MultiGraphics

Version 2.2

Borland Pascal

Turbo Pascal
Program design: Lars Gustafsson

Programming: Lars Gustafsson

Hans Larsson (assembly coding),

Olle Orlander (zooming, SVGA)

Martin Fahlgren (FFE and parts of Pascal translation version 1.0)

Text fonts: Karl Gustafsson (outline) and Lars Gustafsson

Documentation: Lars Gustafsson

English translation: Peter Campbell and Lars Gustafsson

DATABITEN AB

Mail address: Box 115, S-811 22 SANDVIKEN, SWEDEN

Telephone: ++46-26-25 64 93

Telefax: ++46-26-25 36 41

Copyright 1990,1995 by DATABITEN AB. All rights reserved.

MultiGraphics and SuperVisor are trademarks of DATABITEN AB. Other brand and product names are trademarks or registered trademarks of their respective holders.
Introduction
Welcome to the world of **MultiGraphics** - the most extensive graphics toolbox for Borland Pascal/Turbo Pascal ever!

In the **MultiGraphics** manuals you will find detailed descriptions of all units and their constants, data types, variables, procedures and functions, etc, that make up the graphics capabilities of **MultiGraphics**. However, you are not advised to read the manuals in one go, but to keep the manuals as a *reference* close to your computer. The manuals have been arranged and organized on the basis of the units that are included in **MultiGraphics**.

After you have read this introduction, installed the software and read the READ.ME file on one of the distribution disks, we recommend you to take a look at Reference 1, "Let's Draw".

**License conditions**

The license terms for **MultiGraphics** give you the right to use units and drivers from **MultiGraphics** in all applications you use for your own personal use.

You may make copies of the original disks for your own backup purposes, but you are not allowed to make any copies of the printed documentation.

Further, the **MultiGraphics** license gives you the right to freely distribute *compiled applications* (EXE files) that use **MultiGraphics** and its supplied drivers and fonts. You are allowed, moreover, to distributed drivers (files of

---

*MultiGraphics & Borland Pascal/Turbo Pascal*  
DATABITEN
the MGV, MGA, MGH, XGV, XGA and XGH types) and font files with compiled applications.

You are allowed to distribute uncompiled programs or program units that use units and/or drivers in MultiGraphics provided that no units or drivers from MultiGraphics are supplied with them.

The license conditions stated above are binding, even without a signature, and are a condition for the validity of purchase. You are not allowed, on the other hand, to distribute programs that use MultiGraphics or its drivers if you have not signed and returned the license agreement.

The Demo Version of MultiGraphics may only be used for evaluation purposes.

The MultiGraphics Demo Version

The Demo version of MultiGraphics is contained in the compressed file MGTPDEMO.ZIP. Create a new directory, for example C:\MG, and make a copy of MGTPDEMO.ZIP in this directory. Then use the PKUNZIP.EXE utility to uncompress the MultiGraphics demo files.

After unpacking locate the file named

READ.ME

This is a very important text file with information on the Demo version, not mentioned in this documentation. Please read it!

You can now skip the information on the Commercial Version and continue with the "Compiler Recommendations" section later in this introduction.

The MultiGraphics Commercial Disks

MultiGraphics is usually delivered on 3.5" 1.44 MB disks. No matter what the format, the disks contain a number of compressed files, each one of which in turn contains a number of other files. The compressed files must first be unpacked before you can access the individual files in
**Introduction**

**MultiGraphics.** The compressed files have the file type `.EXE' and unpack themselves, i.e., by executing the compressed file, the individual files it contains are created simultaneously.

**READ.ME**

Important text file with the latest news. Read it!

**MGIFC.EXE**

This is a compressed file with interface sections to all Pascal units in MultiGraphics. They are not needed when compiling programs but do add valuable information about each device.

**MGTPUxx.EXE**

MGTPU70.EXE, MGTPU60.EXE etc. are compressed files containing MultiGraphics TPU files for Borland Pascal/Turbo Pascal version x.x and real mode. These TPU files are mandatory when you compile graphics applications for real mode. Unpack the files to your special TPU directory or specific MultiGraphics directory.

You only need to unpack the TPU files for your current Turbo Pascal/Borland Pascal version. Borland Pascal 7.0 and Turbo Pascal 7.0 share the same TPU's in MGTPU70.EXE.

**MGTPPPxx.EXE**

Compressed files containing MultiGraphics TPP files for Borland Pascal version x.x and protected mode. These TPP files are mandatory when you compile graphics applications for protected mode. Unpack the files to your special TPP directory or specific MultiGraphics directory.

| Borland Pascal version 7.01 or later is required! |

**MGDBGUG70.EXE**

Compressed file containing TPU and TPP files (MGErr, MGDebug, MGEv, MGMouse) compiled with heap supervising and events logging activated. These debug versions files can only be used with Turbo Pascal
7.0 and Borland Pascal 7.0. Unpack these files to a separate directory -
different from the directory where the standard MultiGraphics TPU’s and
TPP’s reside! See Appendix "MGDebug - Debugging" for more information.

Also stores the source code of the unit **MGDebug**. You will only need this
if you wish to compile **MGDebug** with debug information.

**MGDRV.EXE**

This is a compressed file containing all the MultiGraphics drivers for real
mode supporting a wide range of graphics devices. You'll need them when
loading drivers dynamically from graphics applications. Unpack the drivers
to your directory with executable files. The drivers have file extensions
".MGA", ".MGH" or ".MGV".

**MGDRVX.EXE**

This is a compressed file containing all the MultiGraphics drivers for pro-
tected mode supporting a wide range of graphics devices. You'll need them
when loading drivers dynamically from graphics applications in protected
mode. Unpack the drivers to your directory with executable files. The drivers have file extensions ".XGA", ".XGH" or ".XGV".

**MGFONTS.EXE**

This is a compressed file with loadable fonts. Unpack to the same directory
as the graphics device drivers.

**MGGA.EXE**

This is a compressed file with all the examples and demo programs
("Graphics Applications"). The example programs are important comple-
ments to the printed documentation and it is strongly recommended that
you unpack and try out these programs. Some examples also require that the
images in MGPCX are unpacked.

**MGPCX.EXE**

This is a compressed file with PCX images. They are used by some exam-
ple programs.
Installation

All compressed files are EXE files and unpack themselves in the current directory when they are executed. Where the compressed files have a version number, you should only unpack those files that correspond to the Borland Pascal/Turbo Pascal version you are using. Here is an example:

Suppose your current directory is C:\BP\UNITS. The MultiGraphics disk is placed in drive A:. By using the commands:

```
A:MGTPU70 <Enter>
A:MGTPP70 <Enter>
```

all files in MGTPU70.EXE and MGTPP70.EXE are unpacked and placed in the C:\BP\UNITS directory.

Compiler Recommendations

MultiGraphics applications must always be compiled with the floating point emulation activated (this is set using the Options Compiler menu or with the compiler directive $E+).

Graphics applications are often very demanding on memory during compilation and execution. If you follow our suggestions given below, you will avoid the risk of frustration:

```
¨ Whenever possible, use the Borland Pascal's BP.EXE environmenth has the advantage of using the whole of RAM.

¨ Set target to protected mode whenever possible. This will give your applications a maximum of accessible memory.

¨ Applications compiled for protected mode must have access to the files DPMI16.OVLd RTM.RESe the Borland Pascal documentation).

If you choose to work with Borland Pascal's or Turbo Pascal's TURBO.EXEompile to disk - not to memory! (set using Alt-C D)

¨ Place the link buffer on disk - not in memory! (set using Alt-O L L)
“Split the library file TURBO.TPL into its individual TPU files. See the
documentation about TPUMOVER in the Borland Pascal and Turbo
Pascal manuals.

“Using EMS memory, frees memory from the program editor.

“Clear memory from unnecessary memory resident programs. Combine
this measure by using a memory manager such as EMM386 in DOS 5.0
or 6.x, 386-MAX from Qualitas or QEMM from QuarterDeck.

The Reference Manuals

The reference manuals are divided into three sections, namely:

“ a Reference section that starts with a short tutorial chapter and then
describes the individual units in MultiGraphics,

“ a Device Driver section that describes the details of all device drivers and
finally,

“ an Appendix with lists of error messages, fonts and documentation on
virtual memory handling and emulation of the Borland Graphics Interface
(BGI), etc.

Each section is divided into several chapters. The Reference chapters are
numbered 1, 2, 3 etc., the Device Driver chapters are numbered D1, D2, D3,
etc. and the Appendices are "numbered" A, B, C, etc..

Each chapter has pages numbered from 1 and upwards. Consequently, the
index looks like the following:

4-8 chapter 4, page 8 in the Reference section,
D4-8 chapter D4, page 8 in the Device Driver section and
D-8 appendix D, page 8 in the Appendix section.

The Borland Pascal/Turbo Pascal version does not include chapter 16 "MG
- Concurrent Graphics Processes", which describes the multi-tasking sup-
port in MultiGraphics when using the TopSpeed Modula-2, C or C++
compilers.
Units

The following units are described in the Reference section:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MG</td>
<td>2 - 16</td>
<td>The basic unit of MultiGraphics</td>
</tr>
<tr>
<td>MGPat</td>
<td>17</td>
<td>Standard patterns</td>
</tr>
<tr>
<td>MGCol</td>
<td>18</td>
<td>Color models, etc.</td>
</tr>
<tr>
<td>MGFont</td>
<td>19</td>
<td>Font conversion</td>
</tr>
<tr>
<td>MGImg</td>
<td>20</td>
<td>Images</td>
</tr>
<tr>
<td>MGInt</td>
<td>21</td>
<td>Integer based world coordinates</td>
</tr>
<tr>
<td>MGGW/MGGW87</td>
<td>22</td>
<td>Floating point based world coordinates</td>
</tr>
<tr>
<td>MGIO/MGIO87</td>
<td>23</td>
<td>Text I/O</td>
</tr>
<tr>
<td>MGDL/MGDL87</td>
<td>24</td>
<td>Display lists</td>
</tr>
<tr>
<td>MGDO</td>
<td>25</td>
<td>Graphics objects</td>
</tr>
<tr>
<td>MGOut</td>
<td>26</td>
<td>Printing support</td>
</tr>
<tr>
<td>MGEv</td>
<td>27</td>
<td>Event handling</td>
</tr>
<tr>
<td>MGClock</td>
<td>28</td>
<td>The clock event support</td>
</tr>
<tr>
<td>MGKbd</td>
<td>29</td>
<td>The keyboard event support</td>
</tr>
<tr>
<td>MGMouse</td>
<td>30</td>
<td>The mouse event support</td>
</tr>
<tr>
<td>MGBez</td>
<td>31</td>
<td>Bézier curve fitting</td>
</tr>
<tr>
<td>MGDebug</td>
<td>App. E</td>
<td>Supervising heap management &amp; events</td>
</tr>
</tbody>
</table>

The units MGW, MGIO and MGDL also have versions more suitable for the 80x87, namely MGGW87, MGIO87 and MGDL87. These units set Real = Double and give superior performance when the 80x87 coprocessor or a 486DX is present, but inferior performance without a coprocessor.

The following low level units are not documented:

- PasSys: A connection unit to the compiler's standard library
- MGDef: Declarations of the internal data types for drivers, fonts, symbols, etc..
- MGLow: Low level interface to many graphics operations

See each respective interface section for the available procedures and functions as well as parameter lists.
Typographic Conventions

In the reference manuals, the following conventions are used:

- *Italic* text is used to emphasize words and when new concepts are introduced.

- **Bold** text is used to indicate names of constants, data types, variables, procedures, functions and units when these occur in text.

- Very important paragraphs are marked with double vertical lines in the left margin with the typeface in italic, e.g.:

  The mouse driver must always be loaded!

- Commands and screen output are shown using an italic bold typeface, e.g.:

  GA1 <Enter>

- Descriptions of procedures and functions are introduced with the procedure or the function name inside a double framed box, e.g.:

  | **MatchForeground** | DL |

  or a single framed box, as follows:

  | **SetDevicePalette** |

  A double framed box flags a routine that we believe will be used more often and a single frame flags a routine which is normally seldom used. This is, of course, our highly subjective opinion!

- Some procedures and functions may be marked "DL" to the right of the caption (see **MatchForeground** above). This indicates that the routine may be recorded or stored in display lists. "DLR" and "DL(R)" also occur - see the chapter "MGDL - Display Lists" for an explanation of this.

- Declarations of constants, data types, variables, functions and procedures are shown in bold typeface:

  \[ \text{P MatchForeground(rgbColor: RGB)}; \]
Program listings are printed in a small fixed-width typeface:

USES MG;
VAR
driver: MG.Device;
BEGIN
MG.AutoDevice(driver);

Abbreviations

When declaring procedures or functions the following abbreviations are used:

P Procedure
F Function

Variable and pointer parameters to **MultiGraphics** procedures and functions are classified according to the following system:

<table>
<thead>
<tr>
<th>Comments</th>
<th>Direction</th>
<th>Parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(<em>O</em>)</td>
<td>OUT parameter</td>
<td>variable</td>
</tr>
<tr>
<td>(<em>IO</em>)</td>
<td>IN/OUT parameter</td>
<td>variable</td>
</tr>
<tr>
<td>(<em>Io</em>)</td>
<td>IN/out parameter</td>
<td>pointer type value/variable</td>
</tr>
</tbody>
</table>

An actual parameter value may be a variable or an expression. An actual variable parameter (**VAR** parameter) must be a variable.

An OUT parameter receives a new value *after* the call and its value before the call is not important. An IN/OUT parameter must have a value before the call and is modified during the call.

An IN/out parameter must have a pointer value (an address) *before* the call and this pointer value (address) will not be changed by the call. On the other hand, the data "pointed to" may be modified.
Reference
The basic unit in MultiGraphics is called MG. All graphics applications that use MultiGraphics must therefore import this unit directly or indirectly. This is how it looks in your programs:

USES MG;

To avoid any uncertainty about the origin of identifiers, we consistently qualify them with their unit name. That is, we write MG.DrawLine instead of DrawLine, etc.

We must also load a driver for the device (monitor, printer, plotter) that we want to use. Drivers for many common monitors, printers etc. are supplied with MultiGraphics. Drivers are recognized by the file type "MGA" for real mode and "XGA" for protected/extended mode. The VGA driver is called "VGA.MGA"/"VGA.XGA" for example, for MCGA it is "MCGA.MGA"/"MCGA.XGA", for SuperVGA it is "SVGA.MGA"/"SVGA.XGA", etc. For SuperVGA, there is also a "SVGA.MGH"/"SVGA.XGH" where "H" means that it has a larger default font compared to "SVGA.MGA"/"SVGA.XGA".

Moreover, slimmed-down driver versions, with the file type "MGV" for real mode only, are also included. These drivers do not support virtual screens.

A specific driver is loaded by calling the procedure MG.LoadDevice, in which case the driver's complete file name must be given:
Real mode:

```
MG.LoadDevice(driver1,'VGA.MGA');
MG.LoadDevice(driver2,'C:\MG\drivers\8514A.MGA');
```

Protected/extended mode:

```
MG.LoadDevice(driver1,'VGA.XGA');
MG.LoadDevice(driver2,'C:\MG\drivers\SVGA.XGA');
```

where `driver1` and `driver2` are variables of the type `MG.Device`. The drivers loaded are linked to each respective variable.

By using `MG.AutoDevice`, a suitable driver can also be detected and loaded automatically. You can rely on the automatic detection for the most popular graphics modes, i.e., CGA, EGA, VGA, MCGA and Hercules. `MG.AutoDevice` selects the mode that gives the best resolution and has the most colors. `MG.AutoDevice` expects the driver files to be present in the current directory.

With SuperVGA, VGA is detected.

```
MG.AutoDevice(driver);
```

Here as well, the driver loaded is linked to a variable of the type `MG.Device`.

The third step is to activate the device using

```
MG.SetDevice(driver1);
MG.SetDevice(driver2);
```

etc. This action is necessary because `MultiGraphics` can have many device drivers loaded simultaneously. `MG.SetDevice` decides which device that will receive the graphics commands from then on.

Before we terminate a graphics application, all graphics devices should be closed. This is done by the call

```
MG.CloseDevice(driver1);
MG.CloseDevice(driver2);
```

etc. This command frees the memory that has been used by the driver and resets the graphics mode (normally to text mode).
Program Outline

This is an outline of a MultiGraphics graphics application with auto detection.

```pascal
VAR
driver: MG.Device;
BEGIN
MG.AutoDevice(driver);
MG.SetDevice(driver);

..Draw graphics..

MG.CloseDevice(driver);
END.
```

Here is an outline of a graphics program (real mode) that loads two driver routines (VGA and MCGA):

```pascal
VAR
vga, mcga: MG.Device;
BEGIN
MG.LoadDevice(vga,'VGA.MGA');
MG.LoadDevice(mcga,'MCGA.MGA');
MG.SetDevice(vga);

..Draw VGA-graphics..

MG.CloseDevice(vga);
MG.SetDevice(mcga);

..Draw MCGA-graphics..

MG.CloseDevice(mcga);
END.
```

In protected/extended mode applications, replace "VGA.MGA" with "VGA.XGA" and "MCGA.MGA" with "MCGA.XGA".

Configuring the Compiler

MultiGraphics applications must always be compiled with the floating point emulation activated (this is set using the Options Compiler menu or
Graphics programs are often very demanding on memory during compilation as well as during execution. First of all, we recommend using Borland Pascal's protected mode environment BP.EXE which can use all of your computer's RAM.

If you run the BP environment directly under DOS and the target is for real mode applications, then you should limit the extended memory that is allocated to the integrated environment by setting the environment variable DPMIMEM. If, for example, you want to give the environment 2 MB of extended memory, you would use the following line:

```
SET DPMIMEM=MAXMEM 2000
```

The command can be placed in AUTOEXEC.BAT. Running under Windows 3.x, the memory is instead controlled by using a PIF file. If you do not take this step the applications will not have access to extended memory when you are executing applications from within the integrated environment.

With the real mode TURBO.EXE environment, we recommend that you take the following steps before you compile and test graphics applications:

- Compile to disk - not memory! (set using Alt-C D)
- Set the link buffer to disk - not memory! (set using Alt-O L L)
- Split TURBO.TPL into separate TPU files. See the documentation about TPUMOVER in the Borland Pascal/Turbo Pascal manuals.
- Use EMS memory. This frees memory in the Pascal editor.
- Clear DOS memory from superfluous memory resident programs. Combine this step with using a good memory manager.
- Large graphics applications are most easily managed if compiled to protected mode. Under real mode they can be managed by using overlay techniques. Most units in MultiGraphics can be "overlayed". The exceptions are MGEv, MGClock, MGKbd, MMouse, SV and VM. We recommend you to increase the default size of the overlay buffer in order to speed up overlay swapping (see OvrSetBuf in the Overlay unit).
The Virtual Memory Manager

**MultiGraphics** can use virtual memory in order to store images, screens, fonts, and symbols etc.

In *protected mode* virtual memory is simply the sum of available RAM memory (conventional plus extended) and available disk memory. The virtual memory is controlled by the units VM and FileVM, or automatically by AutoVM.

In *real mode* virtual memory is the sum of conventional heap memory, extended XMS memory, expanded EMS memory and file memory on disk. The virtual memory is controlled by the units VM, XMSVM, EmsVM and FileVM, or automatically by AutoVM.

When you use the virtual memory manager in real mode the strain on conventional heap memory is reduced and the extra memory can be used for program code or overlays. You will find the detailed documentation on the virtual memory manager in Appendix B.

Now Let's Draw!

It's high time to use **MultiGraphics** for something useful. Our first Graphics Application, GA1, draws a line and writes the classic greeting "Hello world!". You will find the complete program text here and on disk:

```pascal
PROGRAM GA1;
USES MG;
VAR
  driver: MG.Device;
BEGIN
  MG.AutoDevice(driver);
  MG.SetDevice(driver);
  MG.DrawLineTo(100,100);
  MG.DrawStr('Hello world!');
  MG.Pause;
  MG.CloseDevice(driver);
END. (*GA1*)
```

Before you run GA1, check that the MultiGraphics drivers are in the current directory so that AutoDevice can find them.

If you get an error message on screen, it may be that the drivers were not found in the current directory.
Testing Drivers without Recompiling

If you want to write portable graphics that will work on different displays and graphic modes, it is useful to be able to check quickly that the result is what was intended or wanted. This is made easy by MultiGraphics!

Auto detection can be skipped by adding a command line parameter after the program name when you start the program. The command line parameter can also be set in the integrated environment. Here's how you start the program directly from DOS:

Compile the program to an EXE file. Go to the DOS prompt by either quitting the integrated environment, or by opening a DOS shell from the environment.

Make sure that the MultiGraphics drivers (file type ".MGA") are present in the current directory. In order to test the compiled program ("GA1.EXE") in EGA mode, for example, type the following:

```
GA1 EGA <Enter>
```

The driver parameter may, in turn, be followed by a mode number that specifies the desired mode of the driver (if the number is left out, mode 0 is used). This is useful for selecting SuperVGA modes. For example, if you have a TSENG 4000 based graphics card and a monitor that can handle 1024x768 pixels, you can test the program with this graphics mode by typing:

```
GA1 SVGA 3 <Enter>
```

Mode numbers for drivers can be found in the corresponding device driver chapter.

If the parameters are left out, i.e.,

```
GA1 <Enter>
```

auto detection is activated instead.

Command line parameters can also be set directly in the integrated environment, without exiting to DOS.
Program Examples

The MultiGraphics disks contain a large number of programs with many examples of MultiGraphics programming techniques. Most of the example files start with the letters "GA" (for "Graphics Application").

A good introduction for the beginner may be to examine and run through the examples numbered GA1..GA15 which introduce new refinements step by step. After that you can experiment freely among the other examples. Test these larger program examples as well:

**FUNC PLOT** (hardware dependent function plotting).

The test programs normally use auto detection of the graphics mode, but you can test other modes by adding command line parameters after the program name (see above).

On the "Animation on the PC" disk (included free), you will find more examples, especially on graphics animation techniques. Be sure to study and run the "CREATOR" example!

Programming Input Devices

MultiGraphics contains a number of drawing routines that are described in the reference chapters that follow. One very important technique in interactive graphics programming is event handling. Here is a short introduction to this technique - for further details see chapters 27 - 30.

The most common input devices used in event handling are the keyboard, the mouse and the clock. We will start by looking at the different input devices one at a time and use a technique known as polling. After that we will go on to the more difficult task of handling several input devices simultaneously.

At the same time, we will introduce new concepts such as event, event generator and event queue. Event controlled programming has proved to be an important and elegant technique for simplifying the handling of multiple input devices.
Input Devices

The most important input devices for direct interaction with the user on today's PC's are:

- the keyboard and
- the mouse (pointing device)

In the broader sense, we may also regard the system clock, disk drives and hard disks, communication links, sensors, window management, etc, as input devices.

MultiGraphics has direct support for the keyboard, the mouse and the clock. It is also possible to write your own routines to handle other input devices and tie them to the event handler.

Polling

The basic technique to get data from an input device is quite simply to ask the device if it has any data to supply and, if so, to request delivery of that data. It is also possible to skip the question and just let the program wait for the data until the device is able to supply it.

This technique is often used when reading the keyboard. We may request data using MG.KeyPressed and read data with MGIO.RdKbd, MGIO.RdStr and even standard Read/ReadLn, etc.

IF MG.KeyPressed THEN MGIO.RdKbd(key);..

This is an example of the practice "ask first and fetch later", while

MGIO.RdStr(s, SizeOf(s) - 1);

gives an example of "wait for delivery".

Polling allows the receiver, i.e., the program, "to be the boss" of the job. The program decides by itself when it wants to ask the device for data. Polling is recommended when working with single input devices and when input data is received at a low frequency, like an interrupt in an otherwise autonomous program. Part of a polling program could look like this:
Problems arise if several input devices compete simultaneously for attention and insist on supplying input data. In what order should we ask them? How long can we let one input device wait before data is lost? What should we do in order not to leave out an input device? The situation could easily degenerate into the type of chaos that occurs around a check-out during a large sale.

Another problem is how we will be able to wait for data from a device and still let the computer execute something useful in the meantime. Perhaps we want to show measurement data at the same time as we are waiting for keyboard input from the user. The computer's attention must be divided between receiving input data and some other tasks. One solution is to allow the program to alternate between the tasks.

Event Handling

Many applications are by their nature interactive, that is they are constantly in direct touch with the user. The user generates events by pressing a key or a combination of keys on the keyboard, he/she moves the mouse, clicks mouse buttons, holds down mouse buttons, releases mouse buttons etc.

Instead of placing the program in the center of things and seeing the processing of input device data as a secondary issue, it is often better to place the events in the center and allow event generating to control the program. Each event triggers an activity on the part of the program.

But how will events of the most diverse types, which occur at the most irregular points in time, be able to co-operate without chaos arising? As shoppers, we are already acquainted with the solution from the supermarket! Of course, all events must be placed in a queue at "the check-out", i.e., the program. The most recent event must place itself last in the queue. The event first in the queue gets served first!
The program's most important task is now to serve the event queue and to see to it that no event is allowed to wait too long or that the queue becomes so large that events are lost. Here is what the program's main loop may look like in principle:

```
REPEAT
  CASE MGEv.NextEventGen OF
    MGEv.nullGen: (* Do something else in the meantime *)
    eventType1    : MGEv.RdNextEvent; Process
    eventType2    : MGEv.RdNextEvent; Process
    ...
  END;
UNTIL done;
```

Event controlled programming solves the problem of dealing with several input devices simultaneously and prevents events from being lost.

Summary

As we have seen, there are two main techniques along with Borland Pascal/Turbo Pascal to handle data from input devices, namely:

- **polling**: the program has control
- **event controlled**: the events have control

To illustrate this, we will now solve the same programming task using the two methods.

Program Example

The task is to receive and print characters from the keyboard in one window, while the mouse is used for line drawing in another window, but only when the left mouse button is held down. In a third window there will be a happily shining sun and in a fourth window we will show the face of a clock which is updated every second. To the user, the four windows will give the impression of working simultaneously.

To read the time and show the clock we will use the unit MGClock. Here in the manual we will only present a fraction of the code, complete implementations will be found on the MultiGraphics disks - see GAClock1, and GAClock2.
Common to all the solutions is the code to open the four windows, setting the background and foreground colors, patterns and initializing the clock. We refer you to the complete programs for more details.

**GAClock1 - Polling Based Version**

When polling we decide for ourselves *if* and *when* we will ask for and fetch data from the input devices. Because of the repetitive nature of the task, the polling has been placed in a program loop in which we read and process data from the different devices in turn. Here is what the polling loop may look like:

```pascal
before: = maxLongInt;
REPEAT (* Polling loop *)
  now: = MGClock.ClockNow;
  IF (now > before + MGClock.ticksPerSecond) OR
      (now < before) THEN BEGIN
    MGClock.ClockToTime(now,h,m,s);
    MGClock.DrawHands(h,m,s);
    before: = now;
  END;

  IF MG.KeyPressed THEN BEGIN
    MG.SetWin(kbdWin);
    MG.RdKey(key);
    IF (key.wrd = MGIO.keys.esc1.wrd) OR
        (key.wrd = MGIO.keys.esc2.wrd) THEN GOTO loopExit
    ELSE IF key.ch >= ' ' THEN
      MGIO.WrChar(key.ch);
  END;
```

```
One problem when polling is that different devices require different amounts of attention. The least demanding is the keyboard which produces data at a relatively slow rate. The internal system clock is updated 18 times per second (under DOS) while the mouse when it is moved normally generates position data at least 30 times per second. One other difference is that the clock always produces data, while the keyboard and mouse may be "asleep" for shorter or longer periods.

The solution as far as the keyboard is concerned, is to "ask first" (using `MG.KeyPressed`) and "read later" (MG.RdKbd). If no key has been pressed, the program continues to the next device. For the mouse, we can read the position and the status of the buttons as often as we want, without waiting for new mouse data. The clock need only be updated once per second, so it is not necessary to call MGClock.DrawHands on each loop. We therefore allow 18 ticks (MG.ticksPerSecond) to elapse between the calls.

However, it would not be detrimental to call MGClock.DrawHands during each turn of the loop, since this procedure only redraws the hands of the clock if they have been moved. We would however avoid the time it takes to convert from ticks to hours, minutes and seconds.

One weakness in the polling based solution is that we may miss data from the mouse if the processing of the other devices is held up in any way (this does not apply to the mouse pointer which is updated automatically in the background). When we draw with the mouse, the operation may become
jerky because we register the mouse's position too sporadically.

**GAClock2 - Event Based Version**

In the event based version, we allow the event handler in `MGEv` to line up all the events in a queue in strict order of time. The program's task is reduced to that of a doorman, to let the events in for processing and presentation of data in turn. The following shows what the event loop may look like:

```pascal
MGClock.SetRepeatedAlarm(0,MGClock.ticksPerSecond);
done:= FALSE;

REPEAT (* Event loop *)
  CASE MGEv.NextEventGen OF
    MGEv.clockGen:
      BEGIN
      MGEv.RdNextEvent(clockEv,evAge);
      IF (evAge < 10) AND
         (MGClock.repeatedAlarm IN clockEv.events) AND
         (0 IN clockEv.repeatedAlarms) THEN BEGIN
      MGClock.ClockToTime(clockEv.ticks,h,m,s);
      MGClock.DrawHands(h,m,s);
      END;
      (* Draw some graphics *)
      MG.SetWin(sunWin);
      MG.SetPos(0,clockWinHeight);
      MG.SetDir(2730 + 30*Random(29));
      MG.MovePos(radius + 1);
      MG.DrawLine(2*MG.CurWinWidth);
      END;
    MGEv.kbdGen:
      BEGIN
      MGEv.RdNextEvent(kbdEv,evAge);
      IF (evAge < 4) THEN BEGIN
      IF (kbdEv.key.wrd = MGIO.keys.esc1.wrd) OR
         (kbdEv.key.wrd = MGIO.keys.esc2.wrd) THEN
        done:= TRUE
      ELSE IF kbdEv.key.ch >= ' ' THEN BEGIN
      MG.SetWin(kbdWin);
      MGIO.WrChar(kbdEv.key.ch);
      END;
      END;
      END;
    END;
  END;
```

**MultiGraphics & Borland Pascal/Turbo Pascal**

DATABITEN
MGEv.mouseGen:
BEGIN
  MGEv.RdNextEvent(mouseEv,evAge);
  IF (evAge < 2) THEN BEGIN
    MG.SetWin(mouseWin);
    IF MGMouse.leftButtonPressed IN mouseEv.events THEN
      MG.SetPos(mouseEv.scrX - MG.WinX(mouseWin),
                mouseEv.scrY - MG.WinY(mouseWin));
    ELSE IF MGMouse.leftButton IN mouseEv.buttons THEN
      MG.DrawLineTo(mouseEv.scrX - MG.WinX(mouseWin),
                     mouseEv.scrY - MG.WinY(mouseWin));
  END;
END;
END; (*case*)
UNTIL done;

Using MGEv.NextEventGen, we can find out which event generator is responsible for the first event in the queue. If the queue is empty, MGEv.nullGen is returned and nothing is carried out in the loop (we could, on the other hand, have used this non-event to update the sun window).

If there are events in the queue, we can then use the information from MGEv.NextEventGen to select how the event is to be read and processed.

Since the clock only needs to be updated once per second, we can set a repeating "alarm" that generates an alarm event approximately each second. At each tick of the clock, the sun window is updated (on the other hand, we could have placed this updating outside the CASE statement when MGEv.NextEventGen returns MGEv.nullGen).

One advantage with the event controlled solution is that the same model can be used to receive data from all the input devices. Another advantage is that we do not risk losing events from frequent event generators such as the mouse, as long as there is still room in the event queue. The event generators with different speeds automatically receive different amounts of attention in relation to the number of events they generate per time interval. There is, on the other hand, a risk that infrequent event generators must always wait a long time in the event queue before they are processed, especially if the queue is constantly filled by a fast generator.
MG is the central unit in MultiGraphics. In MG, you will find definitions of constants, data types, variables and last but not least procedures that are used by both driver routines and other units in the graphics package, as well as by application programs.

MG includes procedures to:

- load and open graphics devices,
- create and open graphics screens,
- create and open windows,
- change palettes,
- set window modes,
- draw points,
- draw lines, circles, arcs, sectors, ellipses, boxes etc.,
- fill rectangles, circles, sectors, ellipses and other shapes and areas with patterns,
- copy graphics within a window, between windows, between screens and between devices,
- enlarge and reduce graphics (zooming),
- create, save, load, modify, transform and draw symbols,
- create, save, load, modify and transform fonts,
- draw text strings,
- start and stop recording of graphic commands,
- initialize error handling,

Accompanying MG, there are ready-to-use units that extend and enhance these basic capabilities.
All drawing operations in **MG** use device dependent integer coordinates. You will find drawing routines with integer based world coordinates in the **MGIW** unit and floating-point world coordinates are used in the unit **MGW**.

**MGPat** contains hardware independent line styles and patterns and **MGCol** supports more color routines and have several pre-declared colors.

**MG** contains only simple output routines for text strings. For more comprehensive input and output routines, we recommend the **MGIO**, **MGEv**, **MGKbd** and **MGMouse** units. **MGEv**, together with **MGClock**, **MGKbd** and **MGMouse**, is the base for the event handling support in **Multi-Graphics**.

There are routines for image support in the **MGImg** unit.

Using **MGDL** you will be able to record drawing commands and store them in display lists. With **MGDO** you can create libraries of graphic objects.

### Constants

```plaintext
unitName  = 'MG';
copyright = 'DATABITEN';
version   = xxx;
```

This gives the unit name, copyright and version number of **MultiGraphics**.

```plaintext
off = FALSE
on  = TRUE
```

These are synonyms of **FALSE** and **TRUE**.

**ticksPerSecond**

System dependent *floating-point* constant defining the number of clock ticks per second from the computer system clock. With DOS this value is approximately 18.2 ticks/s.
**msPerTick**

System dependent *integer* constant defining a rounded value of the number of milliseconds between two subsequent clock ticks.

---

**System Independent Data Types**

*MultiGraphics* builds upon a number of basic, system independent data types that are declared in *PasSys.IFC*. These types are:

- `Byte`, `Byte2`, `Byte4`, `BitSet8`, `BitSet16`, `BitSet32`, `Char8`, `Char16`, `Integer8`, `Integer16`, `Integer32`, `Word8`, `Word16`, `Word32`.

The types above have a fixed size (number of bits), whatever the environment. Under 16 bits DOS they are declared as follows:

```pascal
Byte       = System.Byte;
Byte2      = Word;       
Byte4      = LongInt;    
BitSet8    = SET OF 0..7;  
BitSet16   = SET OF 0..15; 
BitSet32   = SET OF 0..31; 
Char8      = CHAR;       
Char16     = Word;       
Integer8   = ShortInt;   
Integer16  = INTEGER;   
Integer32  = LongInt;   
Word8      = Byte;       
Word16     = Word;       
Word32     = LongInt;   
```
Real Type in MG

Real

System dependent declaration of the floating point number type. Using Borland Pascal/Turbo Pascal this type is equivalent to 6 bytes Real or 8 bytes Double if using or emulating the math coprocessor. The Real type is used as parameter type in the units MGW, MGIO, etc..

Variables

revision: Word;

Contains the current revision of the implementation.

EP: SV.ErrProc;

This is the local error procedure variable in the unit MG.

bufferPtr: Pointer;
bufferSize: Word;
bufferBusy: BOOLEAN;

bufferPtr defines an internal buffer that is used by several procedures in MG and by other units in MultiGraphics. This buffer should not be exploited for your own use, unless you are absolutely sure that this will not conflict with other procedures. You should only use bufferPtr if the flag bufferBusy is FALSE, and you also need to set this flag to TRUE as long as you are using the buffer.

The buffer is allocated automatically on the initialization of the unit MG and the size is by default 6144 bytes in real mode and 32768 bytes in protected mode. If you need a larger buffer, you must first deallocate the old buffer before you allocate the new one. The allocation must appear before activating graphic devices.
MultiGraphics uses the SuperVisor technique for error handling. In short, what this means is that program execution errors in the library routines can be taken care of in two different ways, namely:

- automatically using *error procedures* or
- manually by using *error checks*.

The technique with error procedures is very simple and easy to use when developing and testing programs. Program execution errors and I/O errors are located automatically and are normally shown on the monitor screen with information about the erroneous unit, procedure and type of error. An error message can look like this:

*Error 1007 ($3EF) in MG.LoadDevice. Continue (Y/N)*

The error message that appears on the screen can instead be sent to a printer or to a file on disk. You can even write error procedures that automatically correct the error and continue with the execution of the program.

Many execution errors are caused by programming errors, e.g., calls using invalid parameters, calling procedures in the wrong order and so on. Such errors can be corrected by reprogramming.

Other execution errors are caused by the program's contacts with the "outside" world, a world of sparse memory resources and ill-behaved
diskettes, harddisks, screens, printers and last but not least the human user himself.

These limitations and I/O errors are more difficult to get rid of. A program that wants to survive must therefore accept reality as it is and incorporate tests that can take care of and correct any possible errors. Therefore, it is often more convenient and appropriate to replace error procedures with manual error checking when each critical procedure call has been made.

Error Codes

Many error codes in MultiGraphics are defined in the MGErr unit and in each respective unit. Error codes are organized as follows:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Error Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MG (MGErr)</td>
<td>1000 - 1099</td>
</tr>
<tr>
<td>MGW</td>
<td>1100 - 1119</td>
</tr>
<tr>
<td>MGIW</td>
<td>1120 - 1149</td>
</tr>
<tr>
<td>MGEv</td>
<td>1150 - 1159</td>
</tr>
<tr>
<td>MGClock</td>
<td>1160 - 1169</td>
</tr>
<tr>
<td>MGKbd</td>
<td>1170 - 1179</td>
</tr>
<tr>
<td>MGMouse</td>
<td>1180 - 1199</td>
</tr>
<tr>
<td>MGIO</td>
<td>1200 - 1249</td>
</tr>
<tr>
<td>MGCol</td>
<td>1250 - 1274</td>
</tr>
<tr>
<td>MGImg</td>
<td>1300 - 1324</td>
</tr>
<tr>
<td>MGFont</td>
<td>1350 - 1374</td>
</tr>
<tr>
<td>MGOut</td>
<td>1425 - 1449</td>
</tr>
<tr>
<td>MGDL</td>
<td>1500 - 1524</td>
</tr>
<tr>
<td>MGDO</td>
<td>1525 - 1549</td>
</tr>
</tbody>
</table>

Device drivers have their own specific error codes. You can recognize them by their large numeric values, always above 10000.

Other supporting units also have their own error codes. The most important are the following:
## Error Handling

<table>
<thead>
<tr>
<th>Unit</th>
<th>Error codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileVM</td>
<td>2 - 160</td>
</tr>
<tr>
<td>EmsVM</td>
<td>128 - 255</td>
</tr>
<tr>
<td>XMSVM</td>
<td>128 - 178</td>
</tr>
<tr>
<td>VM</td>
<td>2200 - 2224</td>
</tr>
<tr>
<td>DBList</td>
<td>2300 - 2324</td>
</tr>
</tbody>
</table>

Finally, error codes are also reported from DOS and BIOS during file i/o and communication. These codes are reported with opposite sign, i.e., their values are < 0.

### Error Procedures

After the call `SV.SetEP(TRUE)`, the error handling in `SV` is automatically set to on, and errors that occur in graphics routines are sent to the current error procedure. The error handling is automatically set to on when `SV` is initialized, as well as indirectly when `MG` is initialized, which means that you normally do not need to call `SetEP`.

### Default Error Handling

As a default, error procedures are used for error handling. When errors occur in graphics drivers, an error message is written in the *current error window*, or in text mode if the graphics has not been activated.

The privilege of dealing with the default error window is allotted to the device that is activated first in the program (using `MG.SetDevice`). The error window is created when the first call is made to `SetDevice`.

The default error window comprises the whole space of the *visible screen* in this device. The current position is assigned to the top left corner in the window. The printout uses the default font in the device.

The error message might look like the following example:

*Error 1007 ($3EF) in MG.LoadDevice. Continue (Y/N)*
SVErrMsg

By importing the unit \texttt{SVErrMsg}, the global output procedure for error messages (\texttt{SV.GlbEP}) is modified to include symbolic error messages instead of just numbers. The text messages are loaded from the file "SVERRMSG.TXT" if it is found in the current directory. This file is a text file with one error message per line.

The decimal error code comprises the first five positions counted from the beginning of each line, followed by a space and the symbolic error message.

\texttt{USES SVErrMsg, MG, MGImg;}

The error messages are written in "plain" English like this

\textit{Error 1007 = driverNotFound in MG.LoadDevice. Continue (Y/N)}

Selecting the Error Window

Any window of the type \texttt{MG.Window} can be used as an error window.

\texttt{SetErrorWin}

\texttt{P SetErrorWin(win: Window);}

This procedure sets the current error window for the program. All processes use this error window, unless the global error procedure \texttt{GlbEP} in \texttt{SV} or the local \texttt{EP} procedures have been redirected.

As a default, an error window is created automatically and installed in the visible screen of the first device opened with \texttt{MG.SetDevice}.

\texttt{SV.Result}
\texttt{invalidWin} invalid window variable

\texttt{VAR}
\hspace{1em}errWin: MG.Window;
\hspace{1em}..
\hspace{1em}MG.CreateWin(errWin,0,0,MG.CurWinWidth,50);
\hspace{1em}MG.SetErrorWin(errWin);
**CurErrorWin**

```pascal
F CurErrorWin: Window;
```

This function returns a reference to the current error window for the program.

## User Defined Error Procedures

To achieve greater flexibility, it is possible to create and install user defined and tailor-made error procedures. With user defined error procedures, errors can be redirected to printers, files on disk and other devices.

## Global Error Procedures

The global error procedure variable `GlbEP` in `SV` controls the error handling of the whole program. Normally, `GlbEP` is linked to a default procedure in `SV`. User defined global error procedures must be of the type `GlbErrProc` in `SV` and must be FAR declared:

```pascal
GlbErrProc = PROCEDURE(unitName: String;
procName: String;
error   : INTEGER);
```

Here is an example of how a user defined global error procedure may be declared:

```pascal
{$F+}
PROCEDURE GlbPrintError(unitName: String;
procName: String;
error   : INTEGER);
```

N.B. the procedure must be declared as **FAR**!

All user defined global error procedures must, as their first statements, include:

```pascal
SV.SetResult(error);
IF NOT SV.UseEP THEN Exit;
```

Moreover, as the last statement you must check to see if tracing is active:

```pascal
IF NOT SV.Trace THEN SV.Terminate;
```
Here is an example of a user defined global error procedure that prints out the error messages on the device \texttt{lst} (e.g., a printer):

\begin{verbatim}
PROCEDURE GlbPrintError(unitName: String;
                         procName: String;
                         error   : INTEGER);
BEGIN
  SV.SetResult(error);
  IF NOT SV.UseEP THEN Exit;
  WriteLn(lst,'Error ',error,' in ',unitName,'.',procName);
  IF NOT SV.Trace THEN SV.Terminate;
END GlbPrintError;
\end{verbatim}

A user defined global error procedure is installed by a call to \texttt{InstallGlbEP} in \texttt{SV}. Using the example above:

\begin{verbatim}
InstallGlbEP(GlbPrintError);
\end{verbatim}

Local Error Procedures in Units

Local error procedures control error handling in individual units and are assigned to the procedure variable \texttt{EP} (each unit has its own \texttt{EP}). As a default, the calls are forwarded to the global error procedure variable \texttt{GlbEP} in \texttt{SV} but you can instead use different local error procedures in each separate unit.

Manual Error Checks

By calling \texttt{SV.SetEP(FALSE)}, the use of automatic error procedures is switched off and any possible error must be checked for manually after each graphics procedure call.

Most procedures and functions in \texttt{MultiGraphics} set a result code during their execution. The result code is an integer. If everything has gone well, the code is set to \texttt{SV.ok} (i.e. 0). When an error occurs, a result code is assigned in accordance with "Error codes" above.
After a call to a graphics procedure, the function `SV.Result` returns the current error code. For example:

```pascal
SV.SetEP(FALSE); (* Error must be checked manually *)
.. MG.LoadDevice(driver,'VGA.MGA');
IF SV.Result <> SV.ok THEN ..
.. MG.CreateMonoVirtScr(scr,200,100);
IF SV.Result <> SV.ok THEN ..
.. SV.SetEP(TRUE);
(* Errors are now taken care of by error procedures *)
```

As the example suggests, the automatic error handling using error procedures can be switched on and off repeatedly during the lifetime of a program. The automatic routine can be used on those sections of the program that have not yet been completely debugged, while manual error checking can be used on sections already debugged.

If manual error checking is used, it is important that you check the result code with `SV.Result` after each call that can lead to an error. Carefully examine each procedure for errors that may arise and then decide if the error, with complete certainty, can be prevented (this is to be preferred) or if an error check is necessary after the call.

For example, the call `MG.SetForeground(color)` reports an error if `color` lies outside the permitted limits for the device color in the current screen. This can be prevented by knowing the maximum value for the device color or by query the maximum permitted value by a call to `MG.CurMaxDeviceColor`.

It is, on the other hand, more difficult to prevent errors that arise when calling memory allocation procedures such as `CreateVirtScr` or I/O based procedures such as `LoadDevice`, `LoadFont`, etc. Unfortunately, it is not within the power of the programmer to banish shortage of memory or malfunctioning diskettes and hard disks. Error checks are therefore absolutely necessary!
The MG unit contains all basic data types and procedures for reading the keyboard. Included are data types for keyboard codes (system specific), types for reading procedures and finally procedure variables for reading single keystrokes and checking the key pressed status.

The keyboard input routines in other MultiGraphics units are based on these data types and elementary operations in MG.

There is also a handy pause procedure ready to use when you want to suspend program execution while waiting for a keystroke. Finally a delay procedure is included to give a timed pause.

You will find more advanced keyboard I/O routines in the MGIO unit and in the event based units MGEv and MGKbd.
Data Types

KbdCode = RECORD
    CASE BOOLEAN OF
        FALSE: (ch : CHAR;
                scan: Byte);
        TRUE : (wrd : Word);
    END;

KbdCode is declared as a variant record. ch is the key's ASCII value. scan is the scan code. wrd combines characters and scan codes to integers.

RdKeyProc = PROCEDURE(VAR key: KbdCode (*O*));

This is the data type for keyboard input procedures. Procedures of this type should return a complete keyboard code of type KbdCode.

KeyPressedProc = FUNCTION: BOOLEAN;

This is the data type for key pressed functions which return TRUE if a key has been pressed, otherwise FALSE is returned.

Procedure variables

RdKey

P RdKey(VAR key: KbdCode (*O*));

This procedure reads individual keystrokes from the keyboard. The procedure waits for the next keystroke.

RdKey is in fact a procedure variable which contains the current keyboard input procedure of RdKeyProc type. As a default, MG.RdKey is assigned the procedure PasSys.RdKey at initialization of the MG unit. Under DOS this procedure reads directly from BIOS.

If the event handler in MGKbd unit is used, RdKey will be redirected to an event based input procedure. This is done automatically at the initialization of the keyboard event generator.
Does not use SuperVisor to set result codes in SV.Result.

```pascal
VAR
  key: MG.KbdCode;
  ..
MG.RdKey(key);
IF key.ch = CHR(27) THEN ..
IF key.scan > 0 THEN ..
IF key.wrd = MGIO.keys.f1.wrd

KeyPressed

F KeyPressed: BOOLEAN;

This function returns TRUE if a key has been pressed, otherwise FALSE.

KeyPressed is in fact a procedure variable which contains the current keyboard check function of KeyPressedProc type. As a default, MGKeyPressed is assigned the function PasSysKeyPressed at initialization of the MG unit. Under DOS this procedure calls BIOS.

If the event handler in MGKbd unit is used, KeyPressed will be redirected to an event based keyboard check function. This is done automatically at the initialization of the keyboard event generator.

Does not use SuperVisor to set result codes in SV.Result.

IF MG.KeyPressed THEN MG.RdKey(key); ..

Procedures

Pause DL

P Pause;

Suspends program execution (or the current process if multi-tasking) until a key has been pressed.

If the keys pressed are Ctrl-C or Esc, the program will be halted by a call to SV.Terminate.

Does not use SuperVisor to set result codes in SV.Result.
Delay

P Delay(ms: Word);

Delays the execution for approximately ms milliseconds (max 65000 ms). This procedure in turn calls SV.Delay.
MultiGraphics has a comprehensive, flexible and easy to use system for handling graphics on monitor screens, dot matrix printers, laser printers, ink jet printers and plotters. The basis for this system is the use of graphic devices.

![Fig 1: Devices in MultiGraphics](image)

Normally, the first task for a program using MultiGraphics is to open one (or more) graphics devices, or simply "devices". The term device is used to...
denote monitor screens, dot matrix printers, laser printers, color printers, plotters and so on. A program can have one or more devices open at the same time. Using multi-tasking, several devices can be run concurrently.

Each device (or rather family of devices) must have its own device *driver*. Drivers can be loaded dynamically during the execution of a program. Using dynamic loading, it is possible to regain memory that has been reserved for a driver and thus use the released memory for other purposes.

A driver can have several *modes* that adapt the driver to different members of a device family or change the resolution and the color capacities of the device.

For example, the VGA driver (640x480 pixels, 16 colors) can also be used in monochrome mode as well as in SuperVGA modes (800x600 pixels, 16 colors). Drivers for Epson printers also have modes for IBM, NEC, as well as for 9 pin and 24 pin printers.

All devices and modes that use a common driver belong to the same *device family*. Each family has a unique *family number* in MultiGraphics.

Before a device can receive graphics commands, it must first be opened and then activated. The process terminates when the device is closed. The main steps are as follows:

1. Open
   - LoadDevice or AutoDevice
2. Output channel
   - SetDeviceOutput
3. Select modes
   - SetDeviceMode,
   - SetOrientation,
   - SetDeviceCall,
   - DeviceCall
4. Activate
   - SetDevice
5. Close
   - CloseDevice

Step 2 is only necessary for printers and plotters. Step 3 is optional.

MultiGraphics supports automatic detection and opening of screen devices. The "basic" device mode is detected with DetectDevice, DetectDeviceDriver or AutoDevice. GetFirstDeviceModeInfo and GetNextDeviceModeInfo present a more complete picture of the device modes supported by the system. You can set the orientation, the aspect ratio, palette and the output routine, as well as use various unique characteristics a device may have such as choosing the resolution, or using
various fonts for printing.

When a device is activated (using SetDevice), a visible screen and a screen window, as large as the visible screen, are automatically created. New windows can be opened in the visible screen or new screens (virtual, virtual monochrome or special), which in turn can have one or more windows, can be created. Windows can also be enhanced by giving them new capabilities.

Dynamic Loading

A driver is loaded dynamically by calling LoadDevice or AutoDevice. These commands open the device as well. Several drivers can be loaded and opened at the same time. Dynamically loaded drivers can be closed and removed from memory by calling CloseDevice.

The standard drivers to be found in MultiGraphics are named after their devices and have the file types ".MGV", "MGA" or "MGH" for real mode applications, and "XGV", "XGA" or "XGH" for extended/protected mode applications, e.g. "CGA.MGV", "VGA.MGA", "EPS.XGA". The "V" means that the driver only supports the visible "screen" (i.e. the paper in a printer or a plotter), "A" (All) means that the driver also supports virtual screens (see "MG - Screens") and copying between screens, while "H" means that the driver has all features of All, but a larger default font for high resolution devices.

PROGRAM Graph;
USES MG;
..  
VAR    
  driver1,driver2: MG.Device;
..  
  MG.LoadDevice(driver1,'VGA.MGA'); (*real mode*)
  MG.LoadDevice(driver2,'HER.MGA');
  MG.SetDevice(driver1);  
  ..  
  MG.SetDevice(driver2);

Change ".MGA" to ".XGA" if the application is compiled for extended/protected mode.

If LoadDevice does not find the specified driver file it will check if a driver is given as the first command line parameter at program start. Such a driver parameter can be followed by a second numerical parameter specifying a desired device driver mode. Here are some examples:
GRAPH <Enter> Use the specified LoadDevice driver only
GRAPH HER.MGA <Enter> Loads HER.MGA for Hercules graphics if the specified LoadDevice driver is not found (real mode)
GRAPH HER.XGA <Enter> Loads HER.XGA for Hercules graphics if the specified LoadDevice driver is not found (protected/extended mode)
GRAPH HER <Enter> Loads HER.MGA/HER.XGA for Hercules graphics if the specified LoadDevice driver is not found
GRAPH SVGA <Enter> Loads SVGA.MGA/SVGA.XGA for SuperVGA if the specified LoadDevice driver is not found
GRAPH svga 2 <Enter> Loads SVGA.MGA/SVGA.XGA for SuperVGA mode 2 if the specified LoadDevice driver is not found

Writing
MG.LoadDevice(driver,'');

will make LoadDevice to only check for a command line parameter driver.

Example using automatic detection:

PROGRAM Graph;
USES MG;
..
VAR
driver: MG.Device;
..
MG.AutoDevice(driver);
MG.SetDevice(driver);

You can override the auto detection mechanism by submitting the program name by a command line parameter specifying a device driver name. This driver parameter can in turn be followed by a second numerical parameter specifying a device driver mode. Examples:

GRAPH <Enter> Auto detection
GRAPH HER.MGA <Enter> Loads HER.MGA for Hercules graphics (real mode)
GRAPH HER.XGA <Enter> Loads HER.XGA for Hercules graphics
The Current Device

Several devices can be open at the same time in an application. The devices are activated using SetDevice.

The order in which the devices are activated is not important except in one particular aspect; in the first device activated, the current error window of the program is created. All error messages from SuperVisor are sent to this window.

One of the open devices becomes the current device (this is set with SetDevice). All procedures that do not have a specific device parameter refer to the current device.

Aspect Ratio

The aspect ratio for a device is used to make corrections for various pixel densities of the width and height of the output media.

The aspect ratio is given in units of parts per thousand. An aspect ratio of 1000 gives the same scale for width and height if the distance between two pixels in the width direction is the same as the distance between two pixels in the height direction. If the distance between two points in width is only 50% of the distance between two points in height, an aspect ratio of 500 should be used.

When a device is opened, a default value is assigned to the aspect ratio. Only exceptionally does this value need to be changed. Small adjustments may be needed, however, in order to compensate for irregularities in different monitors, printers, etc.

The current aspect ratio affects the following procedures:

Constants

driverExt = 'MG';    DOS real mode
driverExt = 'XG';    DOS protected (extended) mode

The first two characters used in the file extension of device drivers, i.e. 'VGA.MGA' (real mode), 'VGA.XGA' (protected mode), 'SVGA.XGH', etc.

getDeviceModeInfo   1
getDeviceSettings   2
detectDeviceMode    3
setSize             5
setDensity          6
setDevicePal        7
setTypeface         8
setTypefaceStyle    9
setTypefaceWeight   10
setTypefaceSize     11
setCharSet          12

The above constants are the standard values for the "n" parameter used when calling the SetDeviceCall and DeviceCall procedures.

Data Types

Device

This is the device data type. You can have several devices active at the same time. Each active device has its own variable of the type Device.
DeviceType = (rasterDevice, 
        vectorDevice, 
        rasterVectorDevice);

This is an enumerated data type to specify the type of the device. In the 
category rasterDevice, you will find the majority of monitor screens, dot 
matrix printers, ink jet printers and laser printers. Pen plotters are to be 
found in the vectorDevice category and in rasterVectorDevice, devices that 
support page description languages such as PostScript.

FileNameString;FileNameStringMG

This string data type is used by DetectDeviceDriver.

DeviceInitProc = PROCEDURE(VAR driver: Device);

This is a procedure type for devices used to define initializing routines that 
are called by OpenDevice. Each device must have at least one initializing 
procedure. Alternative initializing routines may be used in order to optimize 
the size of memory or choice of mode.

DeviceOutputProc = 
    PROCEDURE(    data : Pointer; 
                     count: Word; 
                     VAR error: INTEGER  (*O*));

Procedure type used to define output procedures for certain kinds of 
devices, above all printers and plotters. Output routines are installed by 
calling SetDeviceOutput.

User defined output procedures must save and restore the DS register.

The data parameter is a pointer to a buffer with data that should be sent to 
the device. The number of bytes to be printed is given by count. The error 
variable returns a 0 or an error value.

Ready-to-use output procedures of the type DeviceOutputProc for serial 
and parallel ports and disk files are to be found in the unit MGOut.

Orientation    = (landscape,portrait);
OrientationSet = SET OF Orientation;

Driver routines for certain types of devices, such as printers and pen 
plotters, can orientate the x axis along the long side of the printed surface
(e.g. paper) *i.e.* in *landscape mode* or along the short side (*portrait mode*). The orientation is set (where possible) by calling **SetOrientation**.

The set of possible orientations for a particular device is given by the data type **OrientationSet**. Most monitor screens have the set [landscape].

```pascal
DeviceModeInfo = RECORD
  driverName : FileNameString;
  devFamily  : Word;
  devMode    : Word;
  scrWidth,
  scrHeight  : INTEGER;
  maxColPlane : Word;
  bitsPerPixel: Word;
  trueColor  : BOOLEAN;
END;
```

**DeviceModeInfo** gives the essential information about a detected device's characteristics and suitable driver. Used as a parameter type by **GetFirstDeviceModeInfo** and **GetNextDeviceModeInfo**.

```pascal
DeviceInfo = RECORD
  devFamily    : Word;
  devType      : DeviceType;
  devMode      : Word;
  maxDevMode   : Word;
  scrTypes     : ScreenTypeSet;
  maxColPlane  : Word;
  bitsPerPixel : Word;
  orientations : OrientationSet;
  orient       : Orientation;
  aspect       : INTEGER;
  maxDevPalIdx : Word;
  maxDevPalCol : Word;
  devPalSize   : Word;
  defaultDevPal: DevicePalette;
  maxColor     : DeviceColor;
  maxColMask   : DeviceColor;
END;
```

**DeviceInfo** gives complete information about a device's characteristics and its driver. Used as a parameter type by **GetDeviceInfo**.
nullDevice

Specifies a non-existent device of Device type. Returned when a device driver fails to load.

Procedures & Functions

Device Detection

DetectDevice

P DetectDevice(VAR deviceFamily: Word; (*O*))
        VAR deviceMode : Word (*O*));

This procedure attempts to detect a suitable graphics mode to use on your specific hardware. A suitable family number is returned in the variable deviceFamily and a related device mode in deviceMode.

DetectDeviceDriver in turn calls the system specific procedure MGVESA.GetFirstDeviceModeInfo which can detect the following combinations:

- CGA with color monitor: CGA mode 0. 640x200
- EGA with color monitor: EGA mode 0. 640x350, 16 colors
- EGA with monochrome monitor: EGA mode 1. 640x350, mono
- VGA with color monitor: VGA mode 0. 640x480, 16 colors
- VGA with monochrome monitor: VGA mode 1. 640x480, mono
- MCGA with color monitor: MCGA mode 0. 320x200, 256 colors
- Hercules with mono monitor: Hercules mode 0. 720x350, mono

The default detection is rather conservative and attempts to find a safe base mode. If the computer is equipped with a SuperVGA adapter the detected mode will still be standard VGA 640x480, 16 colors. You can use the procedures GetFirstDeviceModeInfo and GetNextDeviceModeInfo to get a more complete picture of the available modes supported by a particular
configuration.

If there are more than one graphics card in the computer, MG_DetectDevice reports the primary graphics mode. In order to open other graphics cards or other modes or to open devices such as printers and plotters, the drivers must be loaded using LoadDevice, and the required mode set using SetDeviceMode.

For example, if the computer has a SuperVGA card (i.e., one that can handle 800x600 pixels, 16 colors) using a multisync color monitor, plus a Hercules card with an appropriate monochrome monitor and if the color card is the primary card, DetectDevice will choose the standard VGA mode 640x480 pixels, 16 colors as the setting. In order to set the driver to SuperVGA mode, all that is needed is to call SetDeviceMode using a suitable mode number before SetDevice is called. In order to open the Hercules 720x348 card, the Hercules driver must be loaded dynamically using LoadDevice.

For more information, see also CurDeviceFamily and SetDeviceMode.

SV.Result
invalidConfig cannot detect device

**DetectDeviceDriver**

```pascal
PROCEDURE DetectDeviceDriver(VAR driverName: String; (*O*)
  VAR deviceMode: Word (*O*));
```

This procedure attempts to find a suitable graphics driver and mode for your specific hardware. A suitable driver is returned in the variable `driverName` and a related device mode in `deviceMode`.

For more information, see also AutoDevice, CurDeviceFamily and SetDeviceMode.

SV.Result
invalidConfig cannot detect device

VAR
  driver     : MG.Device;
  driverName: MG.FileNameString;
  deviceMode: Word;
..
MG.DetectDeviceDriver(driverName,deviceMode);
MG.LoadDevice(driver,'C:\MG\' + driverName);
MG.SetDeviceMode(driver,deviceMode);
MG.SetDevice(driver);

---

**Ref 5-10**

**MG - Devices**

MultiGraphics & Borland Pascal/Turbo Pascal

DATABITEN
GetFirstDeviceModeInfo

P GetFirstDeviceModeInfo(VAR modeInfo: DeviceModeInfo);

Detects and reports in modeInfo more detailed information on the same graphics mode that is also detected by DetectDevice and DetectDeviceDriver.

This procedure is called when you want to get an overview of all available graphics modes on a specific system and it returns the first available mode. For more modes you have to call GetNextDeviceModeInfo.

SuperVisor error codes are not set.

GetNextDeviceModeInfo

P GetNextDeviceModeInfo(VAR modeInfo: DeviceModeInfo);

Detects additional graphics modes (incl. VESA modes) besides the mode detected by DetectDevice, DetectDeviceDriver and AutoDevice. modeInfo contains detailed information on the detected mode. Each new call returns a new detected mode until modeInfo.devMode is set to 65535.

Even if a specific mode has been detected as possible by GetNextDeviceModeInfo, applications must still check that this mode is accepted when invoking LoadDevice and SetDeviceMode.

Before calls to GetNextDeviceModeInfo you have to first of all call GetFirstDeviceModeInfo. Otherwise modeInfo will contain "garbage" and your application may even crash!

SuperVisor error codes are not set.

VAR
  driver : MG.Device;
  modeInfo: MG.DeviceModeInfo;
  ..
  MG.GetFirstDeviceModeInfo(modeInfo);
  (* Tries to detect a 256 color SuperVGA mode with at least a 640 pixels screen width*)
  WHILE modeInfo.devMode <> 65535 DO BEGIN
    IF (devMode.xRes >= 640) AND
      (devMode.bitsPerPixel = 8) THEN BEGIN
      SV.SetEP(FALSE);
      MG.LoadDevice(driver,devMode.driverName);
      IF SV.Result = SV.ok THEN BEGIN
MG.SetDeviceMode(driver, modeInfo.devMode);
IF SV.Result = SV.ok THEN BEGIN
   SV.SetEP(TRUE);
   MG.SetDevice(driver);
   ...
END;
END;
MG.GetNextDeviceModeInfo(modeInfo);
END; (*while*)
(* Failure *)
..
Some high resolution drivers also have drivers with extension "MGH" (real) or "XGH", e.g. SVGA.MGH, DJET.XGH, etc., where "H" stands for "High". These drivers support larger default fonts than the corresponding "MGA"/"XGA" drivers.

**LoadDevice**, like its alternative **AutoDevice**, only creates the preconditions for using a particular device, it does not set the graphics mode nor make it possible to use the device for drawing. To do this, **SetDevice** must be invoked (see below). Before activating with **SetDevice**, a call can be made to **SetDeviceMode** in order to set an alternative mode in a driver routine (e.g. SuperVGA modes in VGA.MGA/VGA.XGA). The mode is set to 0 as a default.

All devices that are kept open simultaneously must have their own **Device** variables.

In real mode memory used by dynamically loaded drivers is taken from the program heap in conventional memory, even if the virtual memory handler, **VM**, is active.

In protected/extended mode the drivers are loaded as ordinary DLL's ("Dynamic Link Libraries").

A device is closed by a call to **CloseDevice**, and at the same time, the memory reserved for the driver is freed. The device can then be reloaded and reopened by a new call to **LoadDevice**.

**SV.Result**

- **driverNotFound** cannot find the file with the specified driver
- **outOfMem** not enough memory to load the device driver
- **driverNotLoaded** error in loading the device driver
- **invalidDriver** invalid device driver
- **deviceInUse** device already open

VAR

    driver: MG.Device;
    ..
    MG.LoadDevice(driver,'VGA.MGA'); (*reellt läge*)
    ..
    MG.CloseDevice(driver);
    ..
    MG.LoadDevice(driver,'C:\MG\HER.MGV');
**AutoDevice**

```pascal
P AutoDevice(VAR driver: Device (*O*));
```

This procedure automatically detects the type of the graphics adapter that are installed and then dynamically loads a suitable driver for the detected device. The device is then opened (internally using `OpenDevice`) and a suitable mode is set.

The file with the graphics driver must be present in the current directory.

You can override the auto detection mechanism by submitting the program name by a command line parameter specifying a device driver name. This driver parameter can in turn be followed by a second numerical parameter specifying a device driver mode. See "Dynamic Loading" above for examples.

`AutoDevice` calls `DetectDeviceDriver` to detect the graphics adapter in the PC environment. See this procedure and `DetectDevice` for information on which graphics mode that can be detected.

`AutoDevice`, like its alternative `LoadDevice`, only creates the preconditions for using a particular device, it does not set the graphics mode nor make it possible to use the device for drawing. To do this, a call must be made to `SetDevice` (see below). Before activating with `SetDevice`, a call can be made to `SetDeviceMode` in order to set an alternative mode in a driver routine (e.g. Super VGA modes in SVGA.MGA/SVGA.XGA). As a default, the mode is set to a number which is suitable for the hardware detected.

All devices that are kept open simultaneously must have their own `Device` variables. In real mode, memory used by dynamically loaded drivers is always taken from the program heap in conventional memory, even if the virtual memory handler, `VM`, is active. In protected/extended mode the drivers are loaded as ordinary DLL's ("Dynamic Link Libraries").

A device is closed by a call to `CloseDevice`, and at the same time, the memory reserved for the driver is freed. The device can then be re-loaded and re-opened by a new call to `AutoDevice` (or `LoadDevice`).

SV.Result
driverNotFound cannot find the file with the specified driver
outOfMem  not enough memory to load the device driver
driverNotLoaded  error in loading the device driver
invalidDriver  invalid device driver
deviceInUse  device already open

Specific driver errors can also occur when SetDeviceMode is invoked automatically.

VAR
driver: MG.Device;
...
MG.AutoDevice(driver);
MG.SetDevice(driver);

**CloseDevice**

P CloseDevice(VAR driver: Device (*IO*));

This procedure closes a device attached to the variable driver which has previously been opened using OpenDevice, LoadDevice or AutoDevice. In the case of the latter two procedures, the memory which the driver has occupied is returned to the memory pool.

CloseDevice automatically closes all the device's screens, windows and enhanced window capabilities linked to the device's windows. All memory (conventional and virtual) linked to screens, windows and capabilities is freed.

However, CloseDevice does not free memory used by global resources, that is resources not assigned to a particular device. These include loaded symbol libraries, fonts, images, display lists and graphic objects.

If the device to be closed is the current device, the call can only be made if the current window is a screen window in the visible screen (see SetVisiScr and SetScrWin).

If the current device is other than the device to be closed, the call can only be made if the current window is a screen window (see SetScrWin).

SV.Result
onlyInVisiScrWin  only in the visible screen's window
invalidDriver  error in driver parameter
processWithoutWin  calling process not linked to a
invalidScreen
invalidScreenChain
invalidWin
invalidWinChain
capTermError

Errors can also be reported from the virtual memory handler, VM, as well as from the driver's own closing routine.

Device Modes

SetDeviceMode

P SetDeviceMode(driver: Device; (*Io*)
        mode : Word);

This procedure sets the desired mode in the device's driver. All drivers have a standard mode 0 which is automatically activated when opened using OpenDevice or LoadDevice.

Device drivers can have alternative modes (e.g. various resolutions, number of colors, etc) numbered from one (1) upwards. These modes are specific for each device. The highest mode number is given in the field maxDevMode in the record DeviceInfo (see "Data Types" and GetDeviceInfo). The driver's interface section usually exports constants for the various modes (importing these does not normally result in the linking of any extra code).

SetDeviceMode should be called before being activated using SetDevice.

SV.Result
invalidDriver
deviceInUse
invalidDeviceMode

error in the driver
device already activated (using SetDevice)
mode > the highest mode allowed for the driver
The driver routine can also report specific errors when attempts are made to set the mode.

```pascal
USES MG, MGVGA;
VAR
driver: MG.Device;
.. 
MG.LoadDevice(driver,'VGA.MGA'); (*real mode*)
MG.SetDeviceMode(driver, MGVGA.videoSeven800x600x16);
MG.SetDevice(driver);
```

In the example above, a VGA device driver is loaded and set to a mode which is suitable for a Video7 card with a multisync screen (resolution 800x600).

### SetOrientation

```pascal
P SetOrientation(driver: Device; (*Io*)
orient: Orientation);
```

In those cases where it is allowed, this procedure sets the landscape or portrait orientation for the graphics of the device linked to the driver parameter.

**SetOrientation** is often used in conjunction with printers and plotters for choosing horizontal or vertical printouts. However, many devices lack the capability of changing the orientation.

The call must be made before activating the device with **SetDevice**.

**SV.Result**

- **invalidDriver**: error in the driver
- **deviceInUse**: device already activated (using **SetDevice**)
- **invalidOrientation**: the desired orientation does not exist

### SetDeviceOutput

```pascal
P SetDeviceOutput(driver: Device; (*Io*)
Output: DeviceOutputProc);
```

This installs an output procedure, **Output**, for a device driver attached to driver. Output procedures are used mainly by devices such as printers and plotters, with serial or parallel communication, or alternatively for writing to a disk file. Predefined printout routines of this type can be found in the
Output procedures must be installed before the device is activated using SetDevice.

User defined output procedures must save and restore the DS register.

SV.Result

invalidDriver error in driver

USES MG,MGOut;
VAR
  pr: MG.Device;
  ..
 MG.LoadDevice(pr,'HPL.MGA'); (*real mode*)
 MG.SetDeviceOutput (pr,MGOut.COM1);
 MG.SetDevice(pr);

SetDeviceCall

P SetDeviceCall(driver: Device;
  n     : Word;
  par   : Word);

MultiGraphics treats graphics devices in as general a way as is possible. The majority of routines are therefore independent of any particular type of device and as such highly portable.

There are, however, characteristics which are difficult to fit into a general scheme. The procedures SetDeviceCall and DeviceCall offer a way out of these difficulties by making it possible to use specific refinements of a particular device linked to the parameter driver. It goes without saying that such refinements are not portable.

The n parameter provides the call with a function number. There are recommendations for the numbering of device calls in MG (see "Constants" above). These numbers should not be used for other purposes. Among the standard calls are (5) set the size, (6) set the resolution, (7) set the palette, (8) choose a built-in font etc.

SetDeviceCall can be used to set the features of a driver that can be expressed as integers. Therefore the data parameter par is of integer type. If this restriction on input data is too harsh, you may instead use DeviceCall
which accepts any type as an actual parameter. See the respective driver for more information.

**SV.Result** returns the errors that are specific to the call number and the driver.

```pascal
USES MG, MGOut, MGEps;
VAR
  driver : MG.Device;
  ...
  MG.LoadDevice(driver,'EPS.MGA'); (*real mode*)
  MG.SetDeviceOutput(driver,MGOut.LPT1);
  MG.SetDeviceCall(driver, MG.setDensity, MGEps.dens120x72);
```

### DeviceCall

```pascal
P DeviceCall(    driver : Device;
                n      : Word;
                parSize: Word;
                VAR par (*IO*));
```

**MultiGraphics** treats graphics devices in as general a way as is possible. The majority of routines are therefore independent of any particular type of device and as such highly portable.

There are, however, characteristics which are difficult to fit into a general scheme. The procedure **DeviceCall** offers a way out of these difficulties by making it possible to use specific refinements of a particular device linked to the parameter **driver**. It goes without saying that such refinements are not portable.

The **n** parameter provides the call with a number. There are recommendations for the numbering of device calls in **MG** (see "Constants" above). These numbers should not be used for other purposes. Among the standard calls are (1) read the device's mode information, (2) read the current installation information, (5) set the size, (6) set the resolution, (7) set the palette, (8) choose a built-in font etc.

On the other hand, the structure of the untyped variable parameter **par** is totally dependent on the driver and the value of **n**. Structures for the various call numbers are usually defined in the driver's interface section. The **parSize** parameter should equal the byte size of **par**.

**DeviceCall** can be used to set the features of a driver and its device as well
as to read information from the driver. See the respective driver for more information.

**SV.Result** returns the errors that are specific to the call number and the driver.

```pascal
USES MG, MGOut, MGEps;
VAR
driver : MG.Device;
epsInfo: MGEps.DeviceModeInfo;
.. 
MG.LoadDevice(driver,'EPS.MGA'); (*real mode*)
MG.SetDeviceOutput(driver,MGOut.LPT1);
MG.SetDevice(driver);
MG.DeviceCall(driver,
   MG.getDeviceModeInfo, 
   SizeOf(epsInfo),
   epsInfo);
IF MGEps.italic IN epsInfo.typefaceStyles THEN
   MG.SetDeviceCall(driver,
      MG.setToggleStyle, 
      MGEps.italic);
...
```

### GetDeviceInfo

```pascal
PROCEDURE GetDeviceInfo(    driver: Device;
   VAR info  : DeviceInfo (*O*));

This procedure returns information about the driver device in the variable parameter info. See "Data Types" for a description of the DeviceInfo type.

**SV.Result**

**invalidDriver**

error in driver parameter

```pascal
VAR 
driver : MG.Device;
devInfo: MG.DeviceInfo;
..
MG.DeviceInfo(driver,devInfo);
IF devInfo.bitsPerPixel = 1 THEN ..
```
The Current Device

**SetDevice**

 declaring

```pascal
function SetDevice(driver: Device (*Io*));
```

This procedure activates a previously opened device (using **OpenDevice**, **LoadDevice** or **AutoDevice**) attached to **driver** and makes it the current device (for the current process in multi-tasking applications).

The *first* call to **SetDevice**, after the device has been opened, also creates the *visible screen* and its screen window. These then become the current visible screen and current screen window. The window data is set to the same values as for **CreateWin**, i.e. the current position becomes (0,0), etc.

Further calls to **SetDevice** restores the screen and the window that were previously current for the device.

If the device is the *first* to be activated in a program, the current error window located in the visible screen of the current device is also created. All devices in a program share this error window. It is therefore important, first of all, to activate the particular device that is to receive the error messages from **SuperVisor**.

**SV.Result**

- **invalidDriver** error in **driver** parameter
- **outOfMem** not enough memory

**CurDevice**

 declaring

```pascal
function CurDevice: Device;
```

This function returns the current device (for the calling process when multi-tasking).

**SV.Result**

- **processWithoutWin** the process has no window

```pascal
VAR
  oldDriver: MG.Device;
...
  oldDriver := MG.CurDevice;
  MG.SetDevice(driver);
```
.. MG.SetDevice(oldDriver);

**CurDeviceFamily**

```pascal
F CurDeviceFamily: Word;
```

This function returns the *family number* of the current device.

Each device driver has a unique family number which is the same no matter which mode the device has been set for.

```
SV.Result
processWithoutWin the calling process has no window
```

**CurDeviceMode**

```pascal
F CurDeviceMode: Word;
```

This function returns the *mode* for the current device. The same numbering is used here as in the call to **SetDeviceMode**. Each particular device driver has only *one* family number but can have *several* mode numbers.

```
SV.Result
processWithoutWin the calling process has no window
```

**CurOrientation**

```pascal
F CurOrientation: Orientation;
```

This function returns the orientation (portrait or landscape) for the current device.

```
SV.Result
processWithoutWin the calling process has no window
```
**Aspect Ratio**

**SetAspect**

```pascal
P SetAspect(ratio: INTEGER);
```

This procedure sets the aspect ratio for the current device.

Setting the aspect ratio affects all screens and windows for that device. In order to underline this point, the call can only be made from a screen window.

An aspect ratio of 1000 corresponds to a proportion of 1:1, i.e. 1 pixel in the x direction gives 1 pixel in the y direction. Because some devices have different distances between points in the x and y directions, the default aspect ratio is sometimes set to a value of <> 1000. The default aspect ratio is set automatically by `SetDevice`.

```pascal
SV.Result processWithoutWin onlyInScrWin
```

`onlyInScrWin` can only be called from a screen window

```pascal
MG.SetAspect(700);
```

**CurAspect**

```pascal
F CurAspect: INTEGER;
```

This function returns the aspect ratio for the current device. A value of 1000 corresponds to 1:1.

```pascal
SV.Result processWithoutWin
```

`processWithoutWin` the calling process has no window

**Flicker**

**ScanWait**

```pascal
P ScanWait(untilY: INTEGER);
```
This procedure waits until the video memory scanning has reached and bypassed the line with the y coordinate untilY, expressed in the current window coordinates.

The video memory data is read line by line from the top to the bottom of the screen. The data is then sent to the monitor for display. A full screen is read 50 to 80 times each second. When the last line has been displayed, the scan reading restarts from the first line. You will get a more flicker free display if you synchronize your own drawings with this scan process. This is accomplished by waiting until the scan reading has bypassed the line where you want to draw. This will give you approximately 1/50 to 1/80 sec to finish your drawings until the result is displayed on the monitor screen.

Normally, your drawings have an extension in the y direction. Then you have to wait until the scan has reached the lowest point of the drawing, i.e. the point with the lowest y value.

**Drawing operations should be performed without delay after the call of ScanWait.**

Normally, you don’t need to bother with the ScanWait procedure. However, one exception is when you want to display animated, moving symbols or image objects. Then the use of ScanWait is almost mandatory if your goal is a good result.

Calling ScanWait reduces application performance, so you should only use it when it is really necessary.

This procedure does not use the SuperVisor error handler and you cannot test the result by calling SV.Result.

```pascal
MG.SetPos(x,y);
MG.ScanWait(y);
MGImg.DrawImage(img);
```

**WaitNoFlicker**

P WaitNoFlicker;

Waits until the video scan reading has reached the last line in video memory, i.e. y = 0 in screen coordinates.

When the last line in the video memory has been read and displayed on the monitor screen, there is a short delay before the scanning is restarted from the top of the screen. This delay is called the vertical retrace interval. During this retrace interval it is a good choice to complete operations as palette changes.
In the case where the current device is not of CRT type, **WaitNoFlicker** just calls a empty procedure and returns immediately.

*Graphics operations should be performed without delay after the call of **WaitNoFlicker**.*

Calling **WaitNoFlicker** reduces application performance, so you should only use it when it is really necessary.

This procedure does not use the *SuperVisor* error handler and you cannot test the result by calling **SV.Result**.

```pascal
MG.WaitNoFlicker;
MG.SetRGBPalette(rgbPal);
```

**Palette**

See the chapter "MG - Colors".
Each graphics device in MultiGraphics can use one or more screens for drawing. A screen is an integral rectangular drawing area, which can contain one or more windows. Each screen always has a screen window, which normally encloses all of the screen's drawing area. The screen window is created automatically at the same time as the screen.

There are four different screen types in MultiGraphics, namely:

- visible,
- virtual,
- mono virtual and
- special screens.

Each device has one (and only one) visible screen. This screen is created automatically when the device is activated for the first time (using SetDevice). For a monitor, the visible screen consists of the monitor screen, while for a printer or plotter, the paper is the visible screen.

Many devices also support the creation of one or more virtual and/or mono virtual screens. Virtual screens enable you to work with graphics on the quiet, i.e. to do the job without it being displayed. Graphics on virtual screens can be copied at any time to the visible screen or to a file on disk, etc.

The size of virtual screens is not restricted to being the same size as the visible screen, but is allowed to be smaller or larger. Virtual screens can also have other proportions.
Mono virtual screens can only contain monochrome graphics, while virtual screens may contain color information.

Virtual and mono virtual screens, belonging to a particular device, share certain basic features with the visible screen of the same device. This concerns features such as the type of device (raster, vector etc.), the number of bits per pixel in raster devices, orientation of the bits, aspect ratio etc. Virtual screens can, however, differ from the visible screen when it comes to size, number of color planes and the maximum values for color masks and device colors.

Finally, special screens use unique features of the device in order to create further screens. It is not unusual for graphics adapters (e.g. EGA and VGA) to have some extra memory available which can be used for graphics.

All devices must have a visible screen. Other types of screens are not mandatory.
The driver for HPGL plotters, for example, only defines the visible screen, but no virtual screens.

Neither is it mandatory that all drawing routines work for all types of screens.

For example, direct drawing routines for visible screens are not available in the drivers for most raster oriented printer devices. Here all graphics must first be created on a virtual screen and then copied to the visible screen (i.e. the paper).

Graphics can be copied between screens belonging to the same device. Moreover, graphics can sometimes be copied between screens located in different devices. Graphics can also be magnified or reduced (zoomed) when copied between virtual or mono virtual screens.

Virtual and mono virtual screens are created in virtual memory (handled by the VM and AutoVM units). A virtual screen - in a real mode application - can therefore exist in conventional or extended XMS memory, in expanded EMS memory or even on disk depending on what type of memory that was the current when the screen was created. This is important to remember because virtual screens can be very large, especially with color graphics and devices using high resolution, such as laser printers.

**The Current Screen**

A program or a process can have several screens open, but only one screen at any one time is the current screen. Those procedures in MultiGraphics that do not have specific parameters for screens refer to the current screen.

The current screen is set directly with **SetScr** or if it refers to the visible screen with **SetVisiScr**. The current screen can also be set indirectly by calling **SetDevice** or **SetWin**.

**The Visible Screen**

The visible screen is created automatically when a device is activated for the first time (using **SetDevice**). The visible screen refers to the monitor screen when the device is a monitor and the real paper when the device is a printer or a plotter.

A device has one and only one visible screen. The visible screen is attached
to a hidden variable of \texttt{Screen} type.

With monitors and plotters, drawing is often carried out directly on the visible screen. For dot matrix, ink jet and laser printers, it is usually necessary first to draw on a virtual screen and then \textit{copy} the graphics to the visible screen (i.e. the paper).

\section*{Virtual Screens}

A virtual screen is created by calling \texttt{CreateVirtScr}. It is closed by calling \texttt{DisposeScr}. Virtual screens in raster devices use virtual memory to store the color planes data.

The virtual screen can be larger than or smaller than the visible screen. The number of virtual screens that can be open at the same time by a device is limited solely by the amount of virtual memory available (real mode: conventional, extended XMS, expanded EMS or disk).

Virtual screens allow you to perform all types of drawing operations. The number of operations can, however, be limited by the driver's implementation.

It is not mandatory for devices to have the capability of creating virtual screens.

\section*{Max Color Mask}

The virtual screen's maximum color mask is set when the screen is created. The maximum color mask must not be greater than that of the device (i.e. the visible screen's maximum color mask). It can, however, contain fewer device colors (i.e. the color planes in plane oriented devices). This saves memory and makes the drawing and copying operations faster.

The maximum color mask chosen decides which device colors are \textit{allowed} in the screen (see "MG - Window Modes").

The table below shows which device colors are allowed for certain values of the maximum color mask.
<table>
<thead>
<tr>
<th>Max. color mask</th>
<th>Device colors allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0,1</td>
</tr>
<tr>
<td>2</td>
<td>0,2</td>
</tr>
<tr>
<td>3</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>4</td>
<td>0,4</td>
</tr>
<tr>
<td>5</td>
<td>0,1,4,5</td>
</tr>
<tr>
<td>6</td>
<td>0,2,4,6</td>
</tr>
<tr>
<td>7</td>
<td>0,1,2,3,4,5,6,7</td>
</tr>
</tbody>
</table>

Mono Virtual Screens

A mono virtual screen is created by calling `CreateMonoVirtScr`. It is closed by calling `DisposeScr`. Mono virtual screens with raster devices use virtual memory to store graphics.

The mono virtual screen can be larger than or smaller than the visible screen.

The number of mono virtual screens allowed for a device is limited solely by the amount of virtual memory available (real mode: conventional, extended XMS, expanded EMS or disk).

Mono virtual screens allow you to perform all types of drawing operations. The number of operations can, however, be reduced by the driver's implementation. The device colors are also limited to the values 0 and 1.

It is not mandatory for devices to have the capability of creating mono virtual screens.

Special Screens

Certain devices support special screens, by having extra memory available. The features of special screens vary from device to device.

Special screens often produce faster graphics than virtual screens and require less memory because they use the internal memory of the device. A special screen is created using `CreateSpecScr` and is closed with `DisposeScr`.

The capability of creating special screens is not mandatory for devices.
Copying between Screens

Graphics can be copied between screens of the same type and within the same device. Often, graphics can also be copied between screens belonging to different devices. See "MG - Copy Graphics" for details.

Enhanced Screen Capabilities

The capabilities of windows created under a particular screen can be enhanced by calling EnhanceScr. All windows created after this call automatically receive these enhanced capabilities. EnhanceWin does not then need to be called for each individual window.

See "MG - Windows" for a discussion of the concept of "window capabilities".

Data Types

Screen

This is an abstract data type for screens belonging to graphic devices. All screens created are attached to variables of the Screen type. Those screens that are active at the same time must be attached to their own Screen variables. The device’s visible screen is attached, however, to a hidden Screen variable.

ScreenType =
    (visiScr, virtScr, monoVirtScr, specScr);

ScreenTypeSet  = SET OF ScreenType;

ScreenType defines those types of screens that can be created in Multi-Graphics.
**Constants**

nullScreen

Used as a reference to a non-existent screen. Returned when the program fails to create a screen.

\[
\begin{align*}
\text{minScreenType} & = \text{visiScr}; \\
\text{maxScreenType} & = \text{specScr};
\end{align*}
\]

Constants provide the first and last values in the enumerated data type ScreenType.

**Procedures & Functions**

**Creating Screens**

CreateVirtScr

\[
P \text{CreateVirtScr}(\text{VAR scr} : \text{Screen}; (*O*) \\
\quad \text{width, height: INTEGER;} \\
\quad \text{maxColorMask: DeviceColor});
\]

This procedure creates a new virtual screen within the current device and attaches the screen to the \text{scr} variable. The desired screen width, screen height and maximum value for the color mask are specified using the parameters \text{width}, \text{height} and \text{maxColorMask}.

The virtual screen is created in the current virtual memory (installed by \text{VM.SetVM} or automatically by \text{AutoVM}).

The width and height can in principle be chosen from the range of integers (0..32767, for 16-bit integers), but are limited, however, to the amount of memory available and any possible restrictions in the driver's implementation. A common restriction is 16384x16384 pixels.
maxColorMask defines which device colors that can be used in the virtual screen. A virtual screen can either allow all device colors defined by the device (i.e. the visible screen) or else limit the choice. Only device colors contained in maxColorMask are allowed when calling SetForeground, SetBackground, etc. With color plane oriented devices, the amount of memory required for the virtual screen can be reduced by limiting the value of maxColorMask.

If, for example, maxColorMask is set to 12, only the device colors 0, 4, 8 and 12 are allowed.

If maxColorMask is set to 1, only device colors 0 and 1 are allowed. Despite the similarities between this restricted virtual screen and a mono virtual screen, they are still different from each other with regard to copying graphics to other screens.

A virtual screen is closed and the memory reserved is freed by calling DisposeScr.

SV.Result
processWithoutWin the calling process does not have a window
invalidScreenType virtual screens not implemented by the driver
invalidMaxColorMask color mask = 0 or > device's maximum color mask
outOfMem not enough memory
invalidWinSize invalid width and/or height

Other errors can be reported from the VM unit.

VM.SetVM(EmsVM.vmm); (*real mode*)
MG.CreateVirtScr(scr,2000,1000,12);
MG.SetScr(scr);
MG.SetForeground(4);
MG.DrawLineTo(1500,500);
win:= MG.CurWin;
MG.SetVisiScr;
MG.CopyView(win);
CreateMonoVirtScr

P CreateMonoVirtScr(VAR scr : Screen; (*O*)
    width, height: INTEGER);

This procedure creates a new mono virtual screen within the current device
and attaches the screen to the `scr` variable. The desired screen width and
screen height are specified using the parameters `width` and `height`.

The mono virtual screen is created in the current virtual memory.

The width and height can in principle be chosen within the range of integers
(0..32767, for 16-bit integers), but are limited, however, to the amount of
memory available and to any possible restrictions with the driver's
implementation. A common restriction is 16384x16384 pixels.

A mono virtual screen is closed and its reserved memory freed by calling
`DisposeScr`.

SV.Result
processWithoutWin the calling process does not have a
window
invalidScreenType mono virtual screens not implemented
by driver
outOfMem not enough memory
invalidWinSize invalid width and/or height

Other errors can be reported from the VM unit.

MG.CreateMonoVirtScr(scr, 50, 50);
MG.SetScr(scr);
MG.SetPos(25, 25);
MG.FillCircle(25);
win := MG.CurWin;
MG.SetVisiScr;
MG.MatchForeground(MG.red);
MG.MatchBackground(MG.blue);
MG.CopyView(win);
CreateSpecScr

P CreateSpecScr(VAR scr : Screen; (*O*))
  VAR width,height: INTEGER; (*IO*)
  VAR maxColorMask: DeviceColor (*IO*));

This procedure creates, if this is possible, a new special screen within the current device and attaches the screen to the `scr` variable. The desired screen width, screen height and maximum values of the color mask are specified using the `variable` parameters `width`, `height` and `maxColorMask`.

In general, you have to assume that the device chooses its own values for these parameters, the desired values should not therefore be regarded as having been put into effect after the call to `CreateSpecScr` has been made. The values of `width`, `height` and `maxColorMask` can therefore be modified upon return.

Many devices do not implement special screens, while others only allow one special screen having a predefined size and maximum color mask.

Special screens are closed by calling `DisposeScr`.

SV.Result

processWithoutWin the calling process does not have a window
invalidScreenType special screens not implemented by driver
outOfMem not enough memory

CloneScr

P CloneScr( sourceScr: Screen;
            VAR cloneScr : Screen (*O*));

This procedure creates a copy of a virtual or mono virtual screen. The original screen in `sourceScr` is copied to `cloneScr`. The original screen is kept intact in `sourceScr`.

`CloneScr` also installs a screen window corresponding to the original and copies the contents to this.

On the other hand, `CloneScr` does not copy the current window data for the screen window, nor does it install other windows or enhanced window capabilities.
CloneScr is extremely useful if you need to free memory for other screens or resources as images, fonts etc. By changing the current virtual memory - in a real mode application - from conventional memory to extended XMS, expanded EMS or disk, a screen can be copied from conventional memory to extended memory, expanded or disk.

The original screen in sourceScr cannot be a visible screen or a special screen.

SV.Result
processWithoutWin the calling process does not have a window
invalidScreenType invalid type of source screen.
outOfMem not enough memory

Errors can also be reported from the VM unit.

VM.SetVM(XMSVM.vmm);
MG.CloneScr(scr1, scr2);
MG.DisposeScr(scr1);
MG.SetScr(scr2);
win := MG.CurWin;
MG.SetVisiScr;
MG.CopyView(win);

<table>
<thead>
<tr>
<th>ZoomVirtScr</th>
<th>DLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>P ZoomVirtScr(sourceScr: Screen;</td>
<td></td>
</tr>
<tr>
<td>destScr : Screen; (<em>Io</em>)</td>
<td></td>
</tr>
<tr>
<td>mixMode : DrawMode);</td>
<td></td>
</tr>
</tbody>
</table>

The screen content in the virtual screen sourceScr is copied to the virtual screen destScr so that the copy fits exactly into the destScr screen. Depending on the relationship between the widths and heights of destScr and sourceScr respectively, the graphics is enlarged or reduced compared to the original. In general, the ratio between width and height is not conserved.

When graphics is reduced, the merged rows or columns are combined using the drawing mode mixMode. Mostly, the drawing modes replace, mix or mask is used. replace is the fastest, but may lose to much of the information. mix preserves bits that are set to one in the original and therefore tends to make the graphics look "whiter". It is most often used in connection with "white-on-black" graphics. mask preserver bits that are set to zero in the original and therefore tends to make the graphics look "blacker". It is most
often used in connection with monochrome "black-on-white" graphics. Both **mix** and **mask** modes can result in undesirable color changes if the zoomed screen is not monochrome.

The original and the destination screens have to be of the same type, to have the same maximum color mask, the same number of color planes and the maximum device colors. Both screens have to be of virtual or mono virtual type, not of visible or special type.

The original screen is not modified. See also **MGImg.ZoomCopyImage** and **MGImg.ZoomImage**.

**SV.Result**
- **processWithoutWin**: the calling process does not have a window
- **invalidScreen**: invalid original or destination screen
- **invalidScreenType**: invalid type of source screen.
- **invalidDrawMode**: invalid drawing mode.

Errors can also be reported from the VM unit.

```pascal
MG.CreateVirtScr(scr1,320,200,MG.CurMaxColorMask);
MG.CreateVirtScr(scr2,640,480,MG.CurMaxColorMask);
MG.CreateVirtScr(scr3,200,100,MG.CurMaxColorMask);
MG.ZoomVirtScr(scr1,scr2,MG.replace); (* Enlarged *)
MG.SetScr(scr2);
win:= MG.CurWin;
MG.SetVisiScr;
MG.CopyView(win);
MG.ZoomVirtScr(scr1,scr3,MG.mix); (* Reduced *)
MG.SetScr(scr3);
win:= MG.CurWin;
MG.SetVisiScr;
MG.CopyView(win);
```
**DisposeScr**

\[ \text{P DisposeScr(VAR scr: Screen (*IO*));} \]

This procedure closes and erases a virtual, mono virtual or special screen. All windows and window capabilities that belong to the screen are closed and all memory used is returned to the virtual memory handler.

*The current window must be a screen window when the call is made.*

**SV.Result**
- **processWithoutWin**: the calling process does not have a window
- **invalidScreen**: invalid screen variable
- **onlyInScrWin**: only allowed from the screen window
- **invalidWin**: invalid window in the screen's internal chain of windows

Errors can also be reported from the **VM** unit.

---

**Enhancing Screen Capabilities**

**EnhanceScr**

\[ \text{P EnhanceScr( scr: Screen; (*I*) } \]

\[ \text{VAR cap: Capability (*I*));} \]

This procedure increases the capability of all new windows belonging to `scr` that are created hereafter, using the capability that is embodied in the variable `cap`.

In contrast to **EnhanceWin**, the **EnhanceScr** procedure affects all windows that are created afterwards under the screen `scr`.

Those windows in `scr` that have been created previously, e.g. the screen window, do not receive this new capability. Use **EnhanceWin** instead for these.

Still more capabilities can be installed, but a specific capability can only be installed once for a particular screen.

Capabilities are removed in conjunction with the closing of the windows or
The Current Screen

**SetScr**

```pascal
P SetScr(scr: Screen);
```

This procedure makes the screen in `scr` the current screen. The screen window in `scr` then becomes the current window.

The call changes the current device if this is applicable.

**SV.Result**

- **invalidScreen**
  - invalid screen variable in `scr`
- **outOfMem**
  - not enough memory

**CurScr**

```pascal
F CurScr: Screen;
```

This function returns the current screen.

`CurScr` is often used to refer to a hidden screen variable (e.g. the visible screen).

**SV.Result**
processWithoutWin  the calling process does not have a window

When an error occurs, the value nullScreen is returned.

MG.EnhanceScr(MG.CurScr, MGIO.cap);

SetVisiScr

P SetVisiScr;

The visible screen in the current device becomes the current screen. The screen window in the visible screen then becomes the current window.

The procedure makes it possible to access the visible screen which is attached to a hidden Screen variable.

SV.Result
processWithoutWin  the calling process does not have a window
invalidWin  invalid screen window

FlushVisiScr

P FlushVisiScr;

This procedure sees to it that everything drawn on the visible screen is actually printed. The call is used primarily with laser printers, printers with buffers and screens with buffers.

On printers FlushVisiScr produces a form feed.

If the device does not support downloading to buffers, then the call has no effect.

SV.Result
processWithoutWin  the calling process does not have a window

Any possible errors from the driver are reported as well.
CurScrType

F CurScrType: ScreenType;

This function returns the screen type (visible, virtual, mono virtual or special) of the current screen.

SV.Result
processWithoutWin the calling process does not have a window

If an error occurs the value maxScreenType is returned.

IF MG.CurScrType = MG.monoVirtScr THEN ..

CurMaxPlane

F CurMaxPlane: CARDINAL;

This function returns the number for the maximum color plane in the current screen. Color planes are numbered from 0 upwards. For monochrome screens, 0 is returned.

SV.Result
processWithoutWin the calling process does not have a window

CurScrWidth

F CurScrWidth: INTEGER;

This function returns the width of the current screen. For raster devices, the unit is in pixels and for plotters, the length of the plotter unit (the smallest movement of the plotter’s pen) etc.

SV.Result
processWithoutWin the calling process does not have a window

If an error occurs, the value MAXINT is returned.
CurScrHeight

F CurScrHeight: INTEGER;

This function returns the height of the current screen. For raster devices, the unit is in pixels, for plotters the length of the plotter unit (the smallest movement of the plotter's pen) etc.

SV.Result
processWithoutWin the calling process does not have a window

If an error occurs, the value MAXINT is returned.

CurMaxColorMask

F CurMaxColorMask: DeviceColor;

This function returns the maximum color mask value for the current screen.

SV.Result
processWithoutWin the calling process does not have a window

If an error occurs, the value maxDeviceColor is returned.
Each screen in MultiGraphics contains one or more windows. By a window is meant a rectangular drawing area within the screen as well as the accompanying attributes, known as the window mode. All windows belong to a specific screen and a specific device.
Each screen has a screen window, which is created automatically when the screen itself is created. The screen window normally includes the whole screen. The user can create still more windows by calling the procedure `CreateWin`. Such user defined windows are declared as variables of the type `Window`. Each screen holds up to 255 windows.

Window Handles

A window may also be identified by its window handle, which is a number in the range of 0..254 (type `WindowHandle`). All windows owned by a particular screen have their unique window handles. In contrast, windows located in different screens may use the same window handles. The screen window always uses window handle 0, whereas other windows are given handles from 1 and upwards when created. A newly created window always gets the lowest free handle, which implies that when a window is closed and a new one created, the later window will seize the former's handle.

Window handles are useful when you wish to identify the window or windows that contain a specific screen position (`GetWinHandlesAtPos`).

Window Mode

A window has a particular position and size within its screen. The window mode also includes information about the window's current:

- viewport (clipping rectangle),
- foreground and background colors,
- "black-on-white" mode,
- color mask,
- drawing mode,
- drawing position,
- drawing direction,
- line style,
- line width,
- pattern,
- font,
- symbols
- scaling of symbols and
- display list
Positions within a window are given by window coordinates. The window's lower left corner has the coordinates (0,0). The x coordinates increase to the right, while the y coordinates increase upwards. All integer coordinates (−32768..MAXINT) are allowed, even if the position is located outside the window's boundaries (e.g. negative coordinates to the left and below the window).

![Fig 2: Window coordinates](image)

A window's position on the screen is given by its screen window coordinates.

Windows that belong to the same screen can overlap each other. Graphics that are drawn in one window can therefore affect the graphics in other windows. Each window, however, always keeps its own window mode.

Certain features, such as the maximum number of colors, the number of color planes, permitted drawing operations etc., are the same for all windows that belong to a particular screen.
Some features are the same for all screens, and thus all windows, that belong to a particular graphics device. Among these are the current palette, aspect ratio and output routines.

Finally, there are features that are common to all graphics devices. These include the current error window, error procedure.

The Current Window

Graphics in MultiGraphics are window oriented which means that most graphics operations refer to one particular window, known as the current window and its current window mode.

Window oriented graphics operations only affect the current window. The graphics are clipped against the window's boundaries or against the window's viewport, if this is smaller than the window. If the position, color, font etc are changed, they only affect the window mode for the current window. If the program switches to a new window, which then becomes the new current window, the current window mode is automatically changed at the same time.

This makes it easier to keep several different modes active simultaneously. Each "setup" gets its own window (that can, if necessary, overlap other windows). By changing windows, the mode is also changed.

Some functions are supplied in one version referring to the current window and another version where the window is specified as a parameter. This is true for the functions for window handles (CurWinHandle - WinHandle), window position (CurWinX / CurWinY - WinX / WinY) as well as window size (CurWinWidth / CurWinHeight - WinWidth / WinHeight).

Window Capabilities

All windows that are created in MultiGraphics have common basic capabilities which are described above in the section on window modes and in the description of drawing operations in the following chapters. These capabilities can be enhanced by providing the window with new features. Windows with enhanced capabilities still have the same static data types that ordinary windows have, i.e. the type Window. The new capabilities are linked dynamically to the window during program execution.
A window with enhanced capabilities retains the basic capabilities that are common to all windows, and can therefore use all the routines for basic windows. Moreover, it also has new features with its own routines.

New capabilities are linked to a window by calling the procedure **EnhanceWin**. The new capabilities can only be used after this call has been made. On the other hand, all windows that are created in a particular screen can be provided with enhanced capabilities by calling the procedure **EnhanceScr**. Those windows in the current screen that have been created *after* this call, automatically receive the new capabilities.

**MultiGraphics** has standard units that provide windows with capabilities to deal with world coordinates (**MGIW** and **MGW**) and advanced input and output handling of text (**MGIO**). **MGClock** is also using enhanced capabilities as well as the BGI emulation in **MGBGI**.

### Viewport

All graphics that are drawn in a window are clipped against the current *viewport*. When a window is created, the viewport is of the same size as the window.

![Viewport Diagram](image-url)
Data Types

Window

This is the data type for windows in MultiGraphics.

WindowHandle    = Word8;
WindowHandleSet = SET OF WindowHandle;

Data types for window handles (0..255) and sets of window handles.

ModeFlag    = 0..15;
ModeFlagSet = SET OF ModeFlag;

Data types for internal use only.

WindowMode = RECORD
  winX0, winY0          : INTEGER;
  winWidth, winHeight  : INTEGER;
  viewX0, viewY0       : INTEGER;
  viewWidth, viewHeight: INTEGER;
  handle               : WindowHandle;
  foreground           : DeviceColor;
  background           : DeviceColor;
  foregroundRGB        : RGB;
  backgroundRGB        : RGB;
  colMask              : DeviceColor;
  blackOnWhite         : BOOLEAN;
  drwMode              : DrawMode;
  xpos, ypos           : INTEGER;
  angle                : INTEGER;
  lineStyl             : LineStyle;
  lineStylePos         : Word;
  protectLineStyle     : BOOLEAN;
  skipEndPoint         : BOOLEAN;
  lineWidth            : INTEGER;
  pat                   : Pattern;
  fnt                    : Font;
  symbs                : Symbols;
  symbolsScaleX,
  symbolsScaleY        : LongInt;
symbolsAngle         : INTEGER;
symbolsShearX,      
symbolsShearY        : INTEGER;
newModes             : Word;
user                 : Pointer;
END;

Record type for window mode. Used as a parameter type in GetWinMode and SetWinMode.

Capability = RECORD..END;

Data type for capabilities. Used as a parameter type in EnhanceScr and EnhanceWin.

Constants

nullWindow

Specifies a non-existent window of Window type. Returned value when MultiGraphics fails to create a new window.

maxWindowHandle 254

Specifies the largest acceptable window handle.

newView       $01;
newForeground $02;
newBackground $04;
newColMask    $08;
newDrawMode   $10;
newPos        $20;
newAngle      $40;
newLineStyle  $80;
newLineStylePos $100;
newProtectLineStyle $200;
newLineWidth  $400;
newPat        $800;
newFont       $1000;
newSymbols    $2000;
newTransform $4000;
allModeFlags $7FFF;

Constants for internal use only (see type DeviceModeSet).

Procedures & Functions

Creating Windows

CreateWin

\[
P \text{CreateWin} (\text{VAR win} : \text{Window}; (*O*))

\text{scrX0,scrY0} : \text{INTEGER};
\text{width, height} : \text{INTEGER});
\]

This procedure creates a new window in the current screen and device. The window is attached to the window variable \text{win}.

The \text{scrX0,scrY0} parameter provides the desired position for the window's lower left corner given as screen coordinates (screen window coordinates). \text{width} and \text{height} give the width and height of the window. All parts of the window must reside within the current screen. The width and height of the window must also be at least 2 pixels.

When the call is made, the current window (not \text{win}) must be a screen window.

A window creates a new drawing environment with unique data for the current position, viewport, foreground and background colors, color mask, drawing mode, position, line style, line width, pattern, font, symbols and display list. The call to \text{CreateWin}, however, has in itself no visible effect on the screen.

A new window is allowed to overlap previous windows, completely or in part. Thus, the graphics in one window can also affect ("interfere with") previously drawn graphics in other windows.
The new window is initiated with the following window mode (see the data type \texttt{WindowMode}):

- **Viewport position** the window's lower left corner (0,0)
- **Viewport width, height** width of the window, height of the window
- **Device color foreground** true color: device color "white"
- **Device color background** true color: device color "black"
- **RGB color foreground** true color: white
- **RGB color background** true color: black
- **Color mask** max. color mask for the screen
- **Black-on-white** no (\texttt{FALSE})
- **Drawing mode** replace
- **x pos, y pos** the window's lower left corner (0,0)
- **Direction** horizontally to the right (0)
- **Line style** solid (solid)
- **Position in line style** 0
- **Protect line style** off
- **Skip endpoint of lines** no (\texttt{FALSE})
- **Line width** 1
- **Pattern** solid
- **Font** device font
- **Symbols** device symbols
- **X scale symbols** 1:1 (1000)
- **Y scale symbols** 1:1 (1000)
- **Angle symbols** 0
- **X shearing symbols** none (0)
- **Y shearing symbols** none (0)
- **Handle** 0..254
- **New mode** yes
- **User field** \texttt{NIL}

This default mode, however, can be modified by automatically installed screen capabilities.

The capability of a window can be increased by calling \texttt{EnhanceWin}. If the screen which the window belongs to has been previously given increased
capability by a call to \texttt{EnhanceScr}, the new window will inherit this capability.

Windows are closed by calling \texttt{DisposeWin} and indirectly when calling \texttt{DisposeScr} and \texttt{CloseDevice}.

\texttt{SV.Result} \texttt{processWithoutWin} the calling process is not attached to a window \texttt{notScrWin} the current window must be a screen window \texttt{invalidWinSpec} invalid window data \texttt{outOfMem} not enough memory \texttt{capInitError} error when initiating a linked capability

\texttt{VAR} \texttt{scr: MG.Screen;} \texttt{win: MG.Window;}
\texttt{. . . MG.SetScr(scr); MG.CreateWin(win,25,100,200,150); IF SV.Result <> SV.ok THEN . . MG.SetWin(win);}

\textbf{AdjustWin}

\texttt{PROCEDURE AdjustWin(win : Window; (*Io*) scrX0,scrY0 : INTEGER; width,height: INTEGER);}

This procedure adjusts the position and size of previously created window. \texttt{scrX0, scrY0} give the new desired position of the window's lower left corner given as screen coordinates (screen window coordinates). \texttt{width} and \texttt{height} give the window's new width and height. The window as a whole must reside within the current screen. The width and height of the window must be at least 2 pixels.

When the call is made, the current window (not \texttt{win}) must be a \textit{screen window}.

The graphics of the window \texttt{win} will neither be clipped (when its size is changed) nor be moved (when its position is changed). This question must
be solved by the program (perhaps with the help of the display list support or a window capability).

\textbf{SV.Result}
\begin{itemize}
  \item \textbf{processWithoutWin} the calling process is not attached to a window
  \item \textbf{notScrWin} the current window must be a screen window
  \item \textbf{invalidWin} invalid window
  \item \textbf{invalidWinSpec} invalid window data
\end{itemize}

\texttt{MG\textunderscore AdjustWin}\texttt{(win,100,100,400,200)};

\textbf{DisposeWin}

\texttt{P DisposeWin(VAR win: Window (*IO*))};

This procedure closes the window attached to the \texttt{win} variable. It also removes all window capabilities linked to the window \texttt{win}.

When the call is made, the current window must be a screen window.

If the window \texttt{win} is the current error window, then this error window is closed (set to \texttt{nullWindow}). The previous error output routines are reset.

\textbf{SV.Result}
\begin{itemize}
  \item \textbf{processWithoutWin} the calling process is not attached to a window
  \item \textbf{notScrWin} the current window must be a screen window
  \item \textbf{invalidWin} invalid window
  \item \textbf{invalidWinChain} invalid internal chain of windows
  \item \textbf{capTermError} error when terminating window capability
\end{itemize}

\texttt{VAR}
\begin{itemize}
  \item \texttt{win: MG\textunderscore Window;}
  \item \texttt{\ldots}
  \item \texttt{MG\textunderscore CreateWin}\texttt{(win,200,100,100,50)};
  \item \texttt{MG\textunderscore SetWin}\texttt{(win)};
  \item \texttt{\ldots}
  \item \texttt{MG\textunderscore DisposeWin}\texttt{(win)};
\end{itemize}
WinOK

F WinOK(win: Window): BOOLEAN;

This function returns TRUE if the win variable is attached to a window - otherwise FALSE is returned.

IF NOT MG.WinOK(win) THEN MG.CreateWin(win,200,100,100,100);

Window Handles

WinHandle

F WinHandle(win: Window): WindowHandle;

This function returns the handle (0..254) of the win window. This handle is unique for the window in its screen. Windows located in different screens may share the same handle number.

If the window variable is invalid, this function returns 255.

See also CurWinHandle.

SV.Result
invalidWin invalid window variable

VAR
  hndl: MG.WindowHandle;
  ..
  hndl := MG.WinHandle(MG.CurWin);

WinOfHandle

F WinOfHandle(handle: WindowHandle): Window;

This function returns a reference to the window in the current screen which has the specified window handle. If no window has this particular handle, and an error occurs, the nullWindow value is returned.

On each call, WinOfHandle will search the window list of the current screen to find a window with the particular handle. If the screen has numerous open windows and the suitable window happens to be located at the end of the internal list of windows, this
call will be relatively slow. If you frequently need to convert windows to handles, we suggest that you store the handles to be searched for in a look-up table.

SV.Result
processWithoutWin the calling process is not attached to a window

MG.SetWin(MG.WinOfHandle(4));

GetWinHandlesAtPos

P GetWinHandlesAtPos( scrX, scrY :INTEGER;
VAR handles:WindowHandleSet (*O*));

This procedure looks for and returns, in the variable parameter handles, the set of all window handles (0..254) in the current screen whose corresponding windows enclose the given position (scrX, scrY). The position is given in screen coordinates. If this position is not contained in any window of the screen, the empty set is returned.

One application for this may be to locate the mouse pointer.

SV.Result
processWithoutWin the calling process is not attached to a window

When an error occurs, the empty set is returned.

VAR
   buttons : MGMouse.ButtonSet;
   mouseX,mouseY: INTEGER;
   handles   : MG.WindowHandleSet;
   curWinHndl : MG.WindowHandle;
   ..
   curWinHndl:= MG.WinHandle(MG.CurWin);
MGMouse.GetMouseScrInfo(buttons,mouseX,mouseY);
MG.GetWinHandlesAtPos(mouseX,mouseY,handles);
IF curWinHndl IN handles THEN ..
Window Capabilities

EnhanceWin

P EnhanceWin(win: Window; (*Io*)
    cap: Capability);

This procedure enhances the capabilities of the window in win using cap. By enhancing the capabilities of a window, the standard procedures in MG are augmented by new procedures e.g. the handling of world coordinates, text handling etc., depending on the capabilities installed.

All procedures for the data type Window work as before, even if the window variable win can now use new procedures.

By repeated calls to EnhanceWin, a window can be enhanced with even more capabilities.

Capabilities can have their own initializing routines that are called by EnhanceWin. These routines modify, if necessary, the current window mode (position, color, etc) in win.

For example, MGIO.cap moves the current window position to the character position (1,1), i.e. to a position near the window's top left corner.

SV.Result
invalidWin            invalid window variable win
alreadyLinked         capability previously installed (linked)
outOfMem              not enough memory
capInitError          error when initiating capability

USES SV,MG,MGW,MGIO;
VAR
    win: MG.Window;
.. SV.SetUseEP(FALSE);
MG.CreateWin(win,10,10,100,100);
IF SV.Result <> SV.ok THEN ..
MG.SetWin(win); IF SV.Result <> SV.ok THEN ..
MG.EnhanceWin(win,MGW.cap);
IF SV.Result <> SV.ok THEN..
MG.EnhanceWin(win,MGIO.cap);
IF SV.Result <> SV.ok THEN..
MGW.SetPos(0.5,0.5);
MGIO.Wrap('Input name: '); MGIO.Read(s, SizeOf(s) - 1);

Window Data

**WinX**

F WinX(win: Window): INTEGER;

This function returns the x coordinate for the lower left corner of the window attached to the `win` variable.

See also CurWinX.

SV.Result
invalidWin
invalid window variable

IF MG.WinX(txtwin) > 100 THEN ..

**WinY**

F WinY(win: Window): INTEGER;

This function returns the y coordinate for the lower left corner of the window attached to the `win` variable.

See also CurWinY.

SV.Result
invalidWin
invalid window variable

**WinWidth**

F WinWidth(win: Window): INTEGER;

This function returns the width of the window attached to the `win` variable.

See also CurWinWidth.
SV.Result
invalidWin  invalid window variable

When an error occurs, the width of MAXINT is returned.

<table>
<thead>
<tr>
<th>WinHeight</th>
</tr>
</thead>
</table>

F WinHeight(win: Window): INTEGER;

This function returns the height of the window attached to the win variable.

See also CurWinHeight.

SV.Result
invalidWin  invalid window variable

When an error occurs, the height of MAXINT is returned.

The Current Window

<table>
<thead>
<tr>
<th>SetWin</th>
</tr>
</thead>
</table>

P SetWin(win: Window);

This procedure makes the window win the current window.

When activated, the former window mode is restored with regard to the current position, direction, color, font, symbols etc (see the data type WindowMode). By changing windows, you can also quickly change to a new current mode.

SetWin, in general, is a very speedy operation - data does not need to be moved or copied - but instead is done by a simple change of reference.

When the window is activated for the first time, it must be linked to the chain of windows that is attached to the window's screen. Dynamic memory handling is used here, which is somewhat slower.

SV.Result
invalidWin  invalid window variable win
**SetScrWin**

```pascal
P SetScrWin;
```

This procedure makes the screen window in the current screen and device the current window.

In contrast to user defined windows, screen windows do not have an open, accessible `Window` variable and cannot therefore be activated directly using `SetWin` (if not previously attached to a `Window` variable).

**SV.Result**

**processWithoutWin**

the calling process is not attached to a window

```
MG.SetWin(win);
...
MG.SetScrWin;
```

**CurWin**

```pascal
F CurWin: Window;
```

This function returns a reference to the current window in the current screen, device and process.

**CurWin** is especially useful when you want to create references to hidden windows, e.g. a screen window.

**SV.Result**

**processWithoutWin**

the calling process is not attached to a window

When an error occurs, `nullWindow` is returned.

```pascal
MG.SetDevice(dev1);
win:= MG.CurWin;
MG.SetDevice(dev2);
MG.CopyView(win);
```
WinExist

F WinExist(VAR curWin: Window (*O*)): BOOLEAN;

This function returns TRUE if the calling process is attached to a window and returns a reference to the window in the variable parameter curWin. It returns FALSE if the process is not attached to a window and then sets curWin to nullWindow.

The function makes it possible to test if the window exists as well as obtaining a reference to the window in one call.

WinExist does not set the result in the unit SV (SV.Result).

IF MG.WinExist(curWin) THEN..

IsScrWin

F IsScrWin: BOOLEAN;

This function returns TRUE if the current window is a screen window, otherwise FALSE.

Certain procedures, such as closing windows and screens, are only allowed when the current window is a screen window. This can be tested using IsScrWin.

SV.Result
processWithoutWin the calling process is not attached to a window

When an error occurs, FALSE is returned.

IF MG.IsScrWin THEN MG.DisposeWin(win);..

HasCap

F HasCap(cap: Capability): BOOLEAN;

This function tests if the current window has the capability cap. It returns TRUE if this is the case, otherwise FALSE.

SV.Result
processWithoutWin the calling process is not attached to
When an error occurs, **FALSE** is returned.

IF MG.**HasCap**(MGIO.cap) THEN MGIO.SetColRow(10,15);

---

### CurWinHandle

**F CurWinHandle: WindowHandle;**

This function returns the *handle* (0..254) of the *current* window. This handle is unique for the window in its screen. Windows located in different screens may share the same handle number.

See also **WinHandle**.

**SV.Result**

**processWithoutWin** the calling process is not attached to a window

255 is returned on error.

**VAR**

  hndl: MG.WindowHandle;

  ..

  hndl:= MG.**CurWinHandle**;

---

### Window Position and Size

#### CurWinX

**F CurWinX: INTEGER;**

This function returns the x screen coordinate for the lower left corner of the current window.

See also **WinX**.
CurWinY

F CurWinY: INTEGER;
This function returns the y screen coordinate for the lower left corner of the current window.
See also WinY.

CurWinWidth

F CurWinWidth: INTEGER;
This function returns the width of the current window.
See also WinWidth.

CurWinHeight

F CurWinHeight: INTEGER;
This function returns the height of the current window.
See also WinHeight.
SV.Result
processWithoutWin the calling process is not attached to a window

When an error occurs, the width MAXINT is returned.

The Current Window Mode

**SetWinMode**

`P SetWinMode(VAR mode: WindowMode (*I*));`

This procedure sets the window mode for the current window. The `mode` parameter is a record of the type `WindowMode` (see "Data Types" above).

*It is very important that all fields in `mode` are valid, because these fields are not tested in the same stringent way as is the case when each field is set on its own.*

`SetWinMode` is normally used in conjunction with `GetWinMode` in order to save the current window mode and then later restore it.

SV.Result
processWithoutWin the calling process is not attached to a window

```pascal
VAR
  win : MG.Window;
  oldmode: MG.WindowMode;

  MG.SetWin(win);
  MG.GetWinMode(oldmode);
  MG.SetPos(100,150);
  MG.SetForeground(10);

  MG.SetWinMode(oldmode);
```
GetWinMode

\texttt{P GetWinMode(VAR mode: WindowMode (*O*))};

This procedure reads the window mode in the current window and returns the data in the \texttt{mode} record variable parameter of the \texttt{WindowMode} type (see "Data Types" above).

\texttt{GetWinMode} gives an overall picture of the window's current mode. The procedure is often used to save the current mode and then reset it using \texttt{SetWinMode}.

\texttt{SV.Result}
\texttt{processWithoutWin} the calling process is not attached to a window

See \texttt{SetWinMode} for examples.

Clear Window

ClrWin

\texttt{P ClrWin;}

This procedure fills the current window using the current \textit{background color} in \texttt{replace} mode. The full screen color mask is used when filling.

\texttt{SV.Result}
\texttt{processWithoutWin} the calling process is not attached to a window

\texttt{MG.SetBackground(MG.MatchBackground(MG.blue));}
\texttt{MG.ClrWin;}

\footnotesize

Ref 7-22

MultiGraphics & Borland Pascal/Turbo Pascal

DATABITEN
The MG unit has a wide range of procedures for color selection. Our aim has been to satisfy all wishes and requirements when it comes to the handling of color graphics, such as to:

```
° offer a comprehensible system for color selection,
° to provide the maximum possible portability between different devices,
° to provide portability between color devices and monochrome devices,
° use all the color capabilities of the hardware (device) and
° achieve the maximum possible speed when selecting colors.
```

Unfortunately, some requirements conflict with each other (e.g., portability versus speed). In MultiGraphics, this problem has been solved by offering alternative procedures that are optimized for one or several of the above requirements.

Depending on their color capabilities, graphical devices can be divided into two main categories:

```
° true color devices with the capability to display all the colors of nature,
° palette oriented devices with a limited number of simultaneous colors (the palette).
```

The latter category can in turn be divided into two subcategories, namely

```
° devices with various degrees of freedom in the selection of which colors to be included in the palette, and
° devices with a fixed color palette (including monochrome devices).
```
In the last few years, true color devices belonging to the first and ideal category have come onto the market thanks to growing memory capacities, greater speed and declining prices.

**MultiGraphics** supports all these categories with a wide range of routines that span the gap in capabilities between these devices.

The color setting procedures closely mirrors the device capabilities. There are, therefore, two main groups of procedures, namely:

- **true color** oriented, and
- **palette** oriented

The palette oriented procedures contain two subgroups, namely:

- procedures for changing the colors to be included in the palette, and
- procedures for selecting a drawing color (*color index*) from the current palette.

### Color and RGB Values

Physically, a color is a particular wavelength or frequency of visible light. The "degree of color" depends on the intensity of the light. In reality, visible light contains a large number of wavelengths of different intensities. However, the human eye and brain are so constructed that we interpret the multitude of colors as mixtures of just three different color components, namely the primary colors *red*, *green* and *blue*.

By taking our human limitations into consideration, the construction of color monitor screens and graphics adapters are simplified enormously. Using the primary colors, red, green and blue, in different combinations, our eyes (and brains) are tricked into believing that they see reality's very much richer repertoire of wavelengths.

In order to pinpoint a color exactly (considering our eyes' limitations), we can specify the intensities of the color's red, green and blue components. This is called the *RGB*-value of the color. In **MultiGraphics**, these intensities are measured on a scale from 0..999. An RGB value of (0,0,0) corresponds to black, (999,0,0) corresponds to bright red, (0,999,0) to bright green, and (999,999,999) to white etc.

The RGB values correspond almost precisely with the technique used to
operate color monitor screens, which is why we have provided the MG unit with support for specifying and choosing colors using the RGB technique. Apart from simple cases however, it can be difficult to directly translate an RGB value into a subjective impression of a color and vice versa.

Long before the real nature of light and vision were scientifically known, artists worked out a technique that characterized colors with the help of the concepts hue, saturation and value. Many of us will still remember "the color circle" from school.

The technique has been given the name HSV after the initial letters of the three words. In the unit MGCol, you will find procedures that support the HSV technique.

**True Color Devices**

The latest generation of graphical devices have support for direct color specification using color RGB values (and with some intermediate calculations also HSV values). The number of simultaneous colors is practically infinite.

To be classified as a true color device, we usually require that the device supports at least a few hundred distinct color intensities for each primary color. If the device reserves 24 bits (3 bytes) for each pixel, each primary color gets 8 bits (1 byte), which corresponds to 256 intensity levels for red, green and blue each. This is widely accepted as satisfactory for many applications.

One step toward true color displays is devices using 16 bits (2 bytes) for each pixel. 6 bits of red and green each (64 intensity levels) and 4 bits of blue (16 levels) is a common representation.

MultiGraphics supports true color devices with procedures for direct manipulation of RGB colors (MatchForeground, MatchBackground, GetForegroundRGB and GetBackgroundRGB). The current foreground and background colors are individually selected for each window. The MGCol also supports color selection using their HSV values.
Device Colors

Each device uses its own standard for the representation of colors. Colors can be enumerated values or specified RGB values according a given model. **MultiGraphics** has reserved a maximum of four bytes for the representation of these device specific colors, or *device colors*. See the data type **DeviceColor**.

Device colors may be specified directly using **SetForeground** and **SetBackground** and read by calling **CurForeground** and **CurBackground** respectively. With true color devices, the device colors are direct specifications of RGB values, while device colors are used as color numbers if the device is palette oriented (more information below).

Palettes

Many devices have restrictions in the number of colors that can be presented simultaneously. A monochrome device can only show two colors, EGA and VGA can show 16 colors simultaneously, MCGA, SuperVGA and 8514/A can show 256 colors at the same time, many color printers can handle 8 colors, and so on. The human eye, on the other hand, can interpret a broad spectrum of colors.

Today's analog color monitors, however, can already display a considerable selection of color nuances. The restrictions are caused by graphics adapters. Many devices (graphics adapters), therefore, have support for choosing the current combination of colors more freely. So even if a VGA card can only show sixteen colors simultaneously, these sixteen colors can be chosen individually from among $2^{26}$ ($64^3$) nuances!

The current selection of colors is called the current *palette* (bringing us a fragrance from the artist's studio). If we continue to use the VGA as an example, then the palette contains sixteen colors. Which sixteen colors that are used can be chosen rather freely, but when the selection has been made, we must then limit ourselves to drawing using only these sixteen colors (we can change the palette, but more about this later).

The palette's colors can be specified using the RGB values (for maximum portability) or by using the device's specific palette colors (for maximum speed). With the **MGCol** unit, the palette colors can also be specified using HSV values.
**MG - Colors**

**MG** contains procedures for setting both the entire palette and for setting individual palette colors.

The palette influences all screens and all windows belonging to a device. To underline this fact, the palette is only allowed to be changed if the current window is a screen window.

When using graphics on monitor screens, changing to another palette will usually also change the colors of the graphics that has already been drawn. You cannot, therefore, wangle more colors out of the program by switching between different palettes!

Each palette oriented device has its own *current palette*, which is common for all screens and windows that belong to the device.

The current palette can be rapidly set, but in a hardware specific way, using **SetDevicePalette**. It can be made portable using **SetRGBPalette**. Individual palette colors are also set and made portable with **SetRGB**.

Each palette device has a *default palette*, which is set automatically when activated by the device. The default palette can also be set with **SetDefaultPalette**.

**Color Index**

The device colors of a palette oriented device usually specify a color number in the palette, often called a *color index*. The index starts at 0. You can then refer to the palette's colors without having to know which actual palette color is being used.

Monochrome devices have device colors (here color indexes) 0 and 1, EGA and VGA have color indexes 0..15 (EGA/VGA cards also have a border color that is given the number 16), MCGA and 8514/A have color indexes 0..255, etc.

Notice that we differentiate between the term *color*, which corresponds to a physically exact nuance of a color, and the term *device color* (here *color index*), which refers to the numbering of the palette's colors. *Which* color a particular color index corresponds to depends on the current palette.

With many devices, the color index 0 normally corresponds to black while the maximum color index corresponds to white (or "light color"), but this is only valid for the default palette and can be changed to other colors by changing the palette.
The current device color (color index) is selected individually for each window. The device color in one window does not affect the device color in any other window.

![Color palette and color index](image)

Fig 1: Color palette and color index

**Foreground and Background Colors**

Each window has actually two current colors - one color for the *foreground* and one for the *background*. True color devices represent these colors directly using their RGB values or device colors, while palette oriented devices use the corresponding device colors (color indexes).

The foreground color is used for drawing points, lines and those parts of bit patterns that are set to "one", monochrome images, symbols and fonts. The device color for foregrounds is set directly with `SetForeground` (true color devices and palette devices) or as a desired RGB value with `MatchForeground`. The latter sets the closest color (true color devices) or the color in the current palette that is best suited to the desired RGB value (palette devices). Remember that the palette is not modified.

The background color is used for the spaces in dashed and dotted lines and in those parts of bit patterns that are set to "zero", monochrome images, symbols and fonts. The device color for the background is set directly with `SetBackground` or as a desired RGB value with `MatchBackground`. The latter sets the closest color (true color devices) or the color in the current palette that is best suited to the desired RGB value (palette devices). Here as well, remember that the palette is not modified.

To summarize, we have two main alternatives when choosing colors for a
true color device - we may select colors as device dependent device colors or as device independent RGB colors.

With palette oriented devices - we can either change the colors in the palette or we can change the device colors (color indexes) for the foreground and background colors. In the first case, all windows and screens in the current device are affected, but the choice of colors is relatively free. In the other case, only the drawing color in the current window is affected while the choice of colors is more limited.

Black-On-White or White-On-Black

Luminous devices, which produce light by themselves, such as monitor screens, often use a black or dark background while what is drawn has a higher intensity. We call this the "white-on-black" technique.

Devices using reflecting surfaces for representation, such as the paper in printers and plotters, use lower intensities when drawing. Correspondingly, we call this technique "black-on-white". This presentation technique is starting to become more common also on monitor screens. Even if this is controversial, many people maintain that "black-on-white" is less tiring on the eyes.

As the default, MultiGraphics opens all devices (including printers and plotters) using a color (true color devices) or color index (palette devices) suitable for "white-on-black". The foreground color is normally white or a light color, while the background color is black or a dark color.

If instead, you want to use the "black-on-white" technique, you can switch the colors or device colors (color indexes) for the foreground and background colors, that is the device color for the foreground color is set to 0 while the device color for the background color is set to the maximum device color for the screen (using a palette device). On a monitor screen moreover you must first fill the current window with "white".

If you want to use portable color settings between color and monochrome devices, MatchForeground and MatchBackground can cause problems when converting to monochrome screens. As a default, conversion is adapted to suit "white-on-black", which means that MatchForeground translates all colors except black to white, while MatchBackground translates all colors except white to black.
After the call is made to `SetBlackOnWhite(TRUE)`, for monochrome screens only, the `MatchForeground` functions so that all the colors except white are converted to black, while `MatchBackground` converts all colors except black to white. It is recommended that you use this method with the "black-on-white" technique.

The "black-on-white" technique can also be achieved by changing the current palette.

Examples

`GAPal` displays all colors in the default device palette.

Constants

```
maxIntensity          999
```

This specifies the maximum permitted intensity for the data type `Intensity`.

```
maxDeviceColor        4294967295
```

This specifies the max. device color allowed in `MG`. This does not mean that every device will permit this value to be used.
black : RGB = (r:  0; g:  0; b:  0);
blue : RGB = (r:  0; g:  0; b:666);
green : RGB = (r:  0; g:666; b:  0);
cyan : RGB = (r:  0; g:666; b:666);
red : RGB = (r:666; g:  0; b:  0);
magenta : RGB = (r:666; g:  0; b:  0);
brown : RGB = (r:666; g:333; b:  0);
lightGray : RGB = (r:666; g:666; b:666);
darkGray : RGB = (r:333; g:333; b:333);
lightBlue : RGB = (r:333; g:333; b:999);
lightGreen : RGB = (r:333; g:999; b:333);
lightCyan : RGB = (r:333; g:999; b:999);
lightRed : RGB = (r:999; g:333; b:333);
lightMagenta: RGB = (r:999; g:333; b:999);
yellow : RGB = (r:999; g:999; b:333);
white : RGB = (r:999; g:999; b:999);

Pre-declared RGB colors.

nullDevicePalette

The "empty" device palette.

Data Types

Intensity = Word;

This is the data type for specifying the intensity of RGB values. It is declared as a Word, but the only permitted range is between 0..999.

RGB = RECORD
    r,g,b: Intensity;
END;

This is the data type that is used to specify RGB values for colors. See also pre-declared colors under "Constants" above.
RGBArray =  
    ARRAY[0..(maxWord - 5) DIV SizeOf(RGB)] OF RGB;

RGBPaletteStruc = RECORD  
    rgbValues: Word;  
    rgbArr   : RGBArray;  
END;

RGBPalette = ^RGBPaletteStruc;

These are data types for handling RGB palettes.

Basically, an RGB palette is an array of RGB values. Because the number of permitted RGB values depends on the range of the device's palette colors, the unit MG uses a pointer type, RGBPalette, to point to a structure of the type, RGBPaletteStruc. This structure has a field for information about the number of RGB values, rgbValues, and an array of this number of RGB values, rgbArr.

DevicePalette;

This is an abstract data type used to refer to hardware dependent palettes. DevicePalette is a pointer type that can point to different kinds of hardware palettes. The concrete structure of such a palette is usually described in the interface section of the device's driver.

DeviceColor = Word32;

This is the data type for device colors. The actual maximum permitted device color depends on the device and screen.
Procedures & Functions

Palettes

**SetDefaultPalette**

P SetDefaultPalette;

This procedure sets the default palette for the current palette oriented device. The default palette is set automatically when the device is activated with the first `SetDevice` call.

If the device does not support color palettes (true color device) this procedure has no effect.

**SV.Result**

*processWithoutWin* the calling process has no window
*onlyInScrWin* only allowed from the screen window

Errors that are specific for the current device are also reported.

**CurMaxPaletteIndex**

F CurMaxPaletteIndex: Word;

This function returns the maximum index of the palette colors for the current device. This is often the same as the max. device color for the visible screen, but can in certain cases be higher, e.g., EGA/VGA have max. device color = 15, while max. palette index = 16 (the extra index refers to the border frame color).

Returns 0 if the current device does not support palettes (true color device).

**SV.Result**

*processWithoutWin* the calling process has no window

When an error occurs **65535** is returned.

**IF** MG.CurMaxPaletteIndex = 1 **THEN** (* monochrome *)..
RGB Palette

**SetRGBPalette**

```pascal
P SetRGBPalette(rgbPal: RGBPalette);
```

This procedure sets a new palette for the current device. The palette is given in the RGB form. When the call is made, the current window must be a screen window. The device must support color palettes.

*SetRGBPalette* is only allowed to be called if the current window is a *screen window*.

All graphics in the device is affected by the new palette.

**SV.Result**

- **processWithoutWin**: the calling process has no window
- **invalidRGBPalette**: invalid RGB palette in *rgbPal* or palettes not supported
- **onlyInScrWin**: only allowed from the screen window

Errors that are specific for the current device are also reported.

```
TYPE
  RGBArray = ARRAY[0..16] OF MG.RGB;
  VGARGBPalStruc = RECORD
    rgbValues: Word;
    rgbArr   : RGBArray;
  END;
CONST
```
rgbPal: VGARGBPalStruc = (rgbValues: 17;
  rgbArr : ((r:  0;g:  0;b:  0),
            (r: 67;g: 67;b: 67),
            (r:133;g:133;b:133),
            (r:200;g:200;b:200),
            (r:266;g:266;b:266),
            (r:333;g:333;b:333),
            (r:400;g:400;b:400),
            (r:466;g:466;b:466),
            (r:533;g:533;b:533),
            (r:599;g:599;b:599),
            (r:666;g:666;b:666),
            (r:733;g:733;b:733),
            (r:799;g:799;b:799),
            (r:866;g:866;b:866),
            (r:932;g:932;b:932),
            (r:999;g:999;b:999),
            (r:  0;g:  0;b:  0)))

MG.LoadDevice(driver,'VGA.MGA');
MG.SetDevice(driver);
MG.SetRGBPalette(@rgbPal);
FOR colIdx:= 0 TO 15 DO BEGIN
  MG.SetForeground(colIdx);
  MG.FillRect(40*INTEGER(colIdx),100,30,30); (*Grey scale*)
END;

GetRGBPalette

P GetRGBPalette(VAR rgbPal: RGBPalette (*O*));

This procedure reads the current palette and returns this in RGB format.

When the call is made, memory is allocated for the RGB palette and a pointer to this is returned in the variable rgbPal. The memory allocated can be freed by calling DisposeRGBPalette.

Devices without palette support return NIL.

SV.Result

processWithoutWin the calling process has no window
outOfMem not enough memory for palette structure

Errors that are specific for the current device are also reported.

When an error occurs rgbPal is set to NIL.
Here is an example adapted for EGA/VGA that sets the "black-on-white" palette:

```pascal
VAR
  rgbPal: MG.RGBPalette;
  colIdx: MG.DeviceColor;
 ..
  MG.GetRGBPalette(rgbPal);
  rgbPal^.rgbArr[0]:= MG.white;;
  rgbPal^.rgbArr[15]:= MG.black;
  MG.SetRGBPalette(rgbPal);
  MG.DisposeRGBPalette(rgbPal);
FOR colIdx:= 0 TO 15 DO BEGIN
  MG.SetForeground(colIdx);
  MG.FillRect(40*INTEGER(colIdx),100,30,30);
END;
```

### CreateRGBPalette

```pascal
P CreateRGBPalette(rgbValues: Word;
  VAR rgbPal   : RGBPalette(*O*));
```

This procedure allocates memory dynamically for an RGB palette structure that is large enough to hold the palette of the specified `rgbValues` RGB colors. The memory is attached to the palette variable `rgbPal`. The palette structure is returned uninitialized.

The allocated space in memory can be freed by calling `DisposeRGBPalette`.

**SV.Result**

- `outOfMem` not enough memory for palette structure

When an error occurs `rgbPal` is set to `NIL`.

### DisposeRGBPalette

```pascal
P DisposeRGBPalette(VAR rgbPal: RGBPalette (*IO*));
```

This procedure frees memory which has been reserved for the RGB palette attached to the `rgbPal` variable. The variable is set to `NIL`.

**SV.Result**

- `invalidPalette` invalid palette variable

`MG.GetRGBPalette(rgbPal);`
Hardware Palettes

**SetDevicePalette**

```pascal
P SetDevicePalette(devPal: DevicePalette);
```

This procedure sets a hardware specific palette for the current device. The parameter `devPal` must then refer to a palette structure that is compatible with the current device. The palette structure is often defined in the interface section of the device's driver. This palette structure is not normally portable between devices.

The device must support color palettes.

*SetDevicePalette* is only allowed to be called if the current window is a screen window.

All graphics in the device is affected by the new palette.

*SetDevicePalette* is used primarily for speed, often together with *GetDevicePalette*. The current palette is saved using the latter procedure and then reset afterwards with *SetDevicePalette*. One other possible use is to quickly switch between a number of predeclared palettes. This technique can be use when doing animation.

**SV.Result**

- **processWithoutWin**: the calling process has no window
- **onlyInScrWin**: only allowed from the screen window
- **invalidPalette**: invalid palette variable or palettes not supported

Errors that are specific for the current device are also reported.

```pascal
VAR
   pal: MG.DevicePalette;
.. MG.GetDevicePalette(pal);
.. MG.SetDevicePalette(pal);
MG.DisposeDevicePalette(pal);
```
GetDevicePalette

\[ P \text{ GetDevicePalette}(\text{VAR } \text{devPal}: \text{DevicePalette} (*O*))\];

This procedure copies the palette of the current device to a new dynamically allocated area of memory and lets the variable parameter `devPal` refer to this. The palette structure is hardware dependent and is normally defined in the interface section of the device's driver.

If the device does not support color palettes `devPal` contains the value `null-DevicePalette` on return.

The dynamically allocated memory can be freed by calling `DisposeDevicePalette`.

\[ SV.\text{Result} \]
\[ \text{processWithoutWin} \quad \text{the calling process has no window} \]
\[ \text{outOfMem} \quad \text{not enough memory} \]

DisposeDevicePalette

\[ P \text{ DisposeDevicePalette}(\text{VAR } \text{devPal}: \text{DevicePalette} (*IO*))\];

This procedure deallocates the memory space that is attached to the palette variable `devPal`. The size given in the palette variable must agree with the size of the palette in the current device. The palette variable is not allowed to refer to the internal palette that is used by the device (see `CurDevicePalette`).

\[ SV.\text{Result} \]
\[ \text{processWithoutWin} \quad \text{the calling process has no window} \]
\[ \text{invalidPalette} \quad \text{invalid palette variable} \]

CurDevicePalette

\[ F \text{ CurDevicePalette}: \text{DevicePalette}; \]

This function returns a reference to the palette in the current device.

In contrast to `GetDevicePalette`, `CurDevicePalette` does not create a copy of the current palette, but refers directly to this instead. If the reference is saved in a variable and the palette is changed later, the variable will refer to the modified palette.
If the device does not support color palettes, this function returns the value `nullDevicePalette`.

A palette that is returned using `CurDevicePalette` should only be used to inspect the palette values, not to change them.

A reference given by `CurDevicePalette` is not allowed to be used when calling `DisposeDevicePalette`, because this would erase the internal palette for the device.

**SV.Result**

- `processWithoutWin`: the calling process has no window

Errors that are specific for the current device are also reported. When an error occurs, `nullDevicePalette` is returned.

### Individual Palette Colors

#### SetRGB

```pascal
P SetRGB(colIdx : DeviceColor;
        rgbColor: RGB);
```

This procedure sets the palette color for device color (color index) `colIdx` in the current device. The current window must be a screen window and the current device must support color palettes.

The palette color changed, affects all screens and windows in the current device (often even previously drawn graphics).

**SV.Result**

- `processWithoutWin`: the calling process has no window
- `onlyInScrWin`: only allowed from the screen window
- `invalidDeviceColor`: invalid color index or palettes not supported
- `invalidRGB`: invalid RGB value

Errors that are specific for the current device are also reported.

```pascal
MG.SetRGB(1, MG.red);
```

This changes the palette color for device color (color index) 1.
GetRGB

P GetRGB(   color : DeviceColor;
        VAR rgbColor: RGB         (*O*));

This procedure reads and returns the RGB color for the device color color in the current device. If the device is palette oriented, color represents the color index.

If the device is palette oriented, the RGB color read is common for all screens and windows in the current device.

SV.Result
processWithoutWin the calling process has no window
invalidDeviceColor invalid device color

Errors that are specific for the current device are also reported.

VAR
  rgbCol: MG.RGB;
  ..
  MG.GetRGB(4,rgbCol);
  rgbCol.r:= rgbCol.r DIV 2;
  rgbCol.g:= rgbCol.g DIV 2;
  rgbCol.b:= rgbCol.b DIV 2;
  MG.SetRGB(4,rgbCol);

Device Colors & Color Indexes

CurMaxDeviceColor

F CurMaxDeviceColor: DeviceColor;

This function returns the maximum permitted device color (color index if palette oriented) for the current window and screen.

SV.Result
processWithoutWin the calling process has no window

When an error occurs, maxDeviceColor is returned.

VAR
  colIdx: MG.DeviceColor;
  ..
FOR colIdx := 0 TO MG.CurMaxDeviceColor DO..Black-On-White or White-On-Black

**SetBlackOnWhite**

P SetBlackOnWhite(trueOrFalse: BOOLEAN);

This procedure sets the "black-on-white" mode if the parameter is **TRUE** and the "white-on-black" mode if the parameter is **FALSE**.

**CurBlackOnWhite**

F CurBlackOnWhite: BOOLEAN;

This function returns the current value of the "black-on-white" setting in the current window.

When an error occurs, **FALSE** is returned.

IF MG.CurBlackOnWhite THEN BEGIN
   MG.MatchForeground(MG.black);
   MG.MatchBackground(MG.white)
ELSE BEGIN
   MG.MatchForeground(MG.white);
   MG.MatchBackground(MG.black)
END;
Matching Device Colors

MatchingForeground

F MatchingForeground (rgbCol: RGB): DeviceColor;

This function returns the device color whose RGB color best corresponds to the desired RGB foreground color in rgbCol.

If the device is palette oriented, the call does not change the current palette setting, but instead seeks out a palette color in the palette of the current device which, as far as is possible, corresponds to the desired color. For monochrome screens, the choice depends on the "black-on-white" setting. See "Black-On-White or White-On-Black" above.

SV.Result
processWithoutWin the calling process has no window

Errors that are specific for the current device are also reported. When an error occurs maxDeviceColor is returned.

VAR
  fgColor: MG.DeviceColor;
  ..
  fgColor := MG.MatchingForeground (MG.magenta);

MatchingBackground

F MatchingBackground (rgbCol: RGB): DeviceColor;

This function returns the device color whose RGB color best corresponds to the desired RGB background color in rgbCol.

If the device is palette oriented, the call does not change the current palette setting, but instead seeks out a palette color in the palette of the current device which, as far as is possible, corresponds to the desired color. For monochrome screens, the choice depends on the "black-on-white" setting. See "Black-On-White or White-On-Black" above.

SV.Result
processWithoutWin the calling process has no window

Errors that are specific for the current device are also reported. When an
error occurs **maxDeviceColor** is returned.

VAR
  bgColor: MG.DeviceColor;
  ..
  bgColor:= MG.**MatchingBackground**(MG.magenta);

Foreground Colors

<table>
<thead>
<tr>
<th>SetForeground</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SetForeground</strong>(color: DeviceColor);</td>
<td></td>
</tr>
</tbody>
</table>

This procedure sets the device color for the foreground color in the current window. The choice affects only the current window. If the device is palette oriented, the palette is not changed.

**SV.Result**

- **processWithoutWin** the calling process has no window
- **invalidDeviceColor** invalid device color

<table>
<thead>
<tr>
<th>Curforeground</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Curforeground</strong>: DeviceColor;</td>
<td></td>
</tr>
</tbody>
</table>

This function returns the device color value for the foreground color in the current window.

**SV.Result**

- **processWithoutWin** the calling process has no window

When an error occurs, **maxDeviceColor** is returned.
MatchForeground

P MatchForeground(rgbColor: RGB);

This procedure sets the color (true color devices) or color index (palette devices) for the foreground color in the current window whose representation best corresponds to the desired RGB color.

The call does not change the current palette setting of a palette oriented device, but instead seeks out a palette color in the palette of the current device which, as far as is possible, corresponds to the desired color. For monochrome screens, the choice depends on the "black-on-white" setting. See "Black-On-White or White-On-Black" above.

SV.Result
processWithoutWin       the calling process has no window

Errors that are specific for the current device are also reported.

MG.MatchForeground(MG.cyan); MG.DrawLineTo(100,100);

GetForegroundRGB

P GetForegroundRGB(VAR rgbColor: RGB (*O*));

This procedure reads and returns the current foreground RGB color used in the current window. The RGB value is returned in the variable rgbColor.

SV.Result
processWithoutWin       the calling process has no window

Errors that are specific for the current device are also reported.

MG.GetForegroundRGB(rgbColor);
MG.MatchForeground(MG.red);
..  
MG.MatchForeground(rgbColor);
Background Colors

**SetBackgroundColor**

P SetBackgroundColor(color: DeviceColor);

This procedure sets the device color for the background color in the current window. The choice affects only the current window. If the device is palette oriented, the palette is not changed.

SV.Result
- processWithoutWin: the calling process has no window
- invalidDeviceColor: invalid device color

**CurBackgroundColor**

F CurBackgroundColor: DeviceColor;

This function returns the device color for the background color in the current window.

SV.Result
- processWithoutWin: the calling process has no window

When an error occurs maxDeviceColor is returned.

**MatchBackgroundColor**

P MatchBackgroundColor(rgbColor: RGB);

This procedure sets the color (true color devices) or color index (palette devices) for the background color in the current window whose representation best corresponds to the desired RGB color.

The call does not change the current palette setting of a palette oriented device, but instead seeks out a palette color in the palette of the current device which, as far as is possible, corresponds to the desired color. For monochrome screens, the choice depends on the "black-on-white" setting. See "Black-On-White or White-On-Black" above.
Errors that are specific for the current device are also reported.

MG.MatchForeground(MG.white); MG.MatchBackground(MG.blue);
MG.DrawStr('Menu');
In this section, we will deal with certain concepts within the term window mode (see "MG - Window" for a complete description of the "window mode" concept), namely the current

- position,
- direction,
- viewport,
- color mask and
- drawing mode.

Each window has its own current setup for position, direction, viewport, color mask and drawing mode. When you change from one window to another, the mode is also changed. Most drawing operations refer to the current position and in several cases to the current direction as well. The current viewport restricts what is to be drawn in space, while the color mask limits which device colors are to be let through to the screen. The current drawing mode decides how the objects drawn are to be combined with graphics already on the screen.

Color selection is detailed in "MG - Colors" and "MGCol - Hue, Saturation, Value".

Position

The current position is taken as the starting point for many drawing opera-
tions. The position is given in window coordinates, where the window's lower left corner has the coordinates (0,0). The x coordinates increase to the right while the y coordinates increase upwards.

![Diagram of window coordinates](image)

Fig 1: Window coordinates

The current position need not reside within the window, instead the whole of the integer range is allowed to be used (i.e. -32768..32767 if two-bytes integers are used).

When a window is created, the current position is assigned to (0,0), i.e. the window's lower, left corner (this is perhaps modified by automatically linked screen capabilities).

The current position can be set absolutely using `SetPos` and relatively with `SetRelPos` and `MovePos`. The position can be read using `GetPos`, `CurX` and `CurY`. Several other drawing operations affect the current position as well.
Direction

The drawing direction is measured in tenths of a degree (decidegrees). There are 3600 decidegrees in one complete revolution. The direction is measured anti-clockwise using as a starting point the x axis' positive direction (i.e. straight to the right). Directions are always normalized within the range of 0..3599 decidegrees.

![Drawing direction diagram]

**Fig 2: Current drawing direction**

The current direction affects moving operations (MovePos), line drawings (DrawLine), arcs (DrawArc, DrawEllipseArc), sectors (DrawSector, DrawEllipseSector, FillSector), ellipses (DrawEllipse, FillEllipse) and text output (DrawStr).

When a window is created, the current direction is set to 0 decidegrees, i.e. horizontally, straight to the right (possibly modified by an automatically linked screen capability).

The current direction can be set absolutely with SetDir and relatively using TurnDir. The current direction can be read using CurDir.

Viewport

When a window is created, everything that is drawn will be clipped against the window's boundaries. By reducing and moving the window's current viewport, graphics can be clipped against the boundaries of a rectangle that is smaller than the window itself.
N.B. that the viewport only limits the extent of that which is *drawn*. Neither the viewport nor the window restrict the accessible x and y positions; the full integer range (including negative coordinates) is accepted.

For example: if a line has its endpoint outside the current viewport, then only that part of the line which is visible within the viewport will be drawn. The current position is, however, still the line's endpoint, even if this lies outside the viewport or the window.

The current viewport is set with `SetView` and can be read using `GetView`. When a window is created, the viewport is of the same size as the window (possibly modified by an automatically linked screen capability).

Viewports have still more functions: A viewport can be *filled* with a pattern using the procedure `FillView`. The contents of a viewport can be *copied* within the window or to another window with the help of `CopyView`.

*MultiGraphics* can use hardware support for clipping, if this is supported by the device.
Color Mask

The current color mask in a window is used to restrict which device colors (= color indexes with palette oriented devices) that are actually allowed to be used when drawing operations are performed. With plane oriented devices the color mask limits which color planes that are modified during drawing operations and vice versa, which color planes that are protected against being modified.

The color mask acts as a final inspector that stops unauthorized device colors from gaining access to the screen.

When a window is created, the color mask is set to the maximum color mask of the screen (see "MG - Screens"). This means normally that all device colors <= the color mask are let through. All color planes can be used.

Masking works in such a way that the color mask, which is of the type DeviceColor, and device colors are combined using the logical bit operation AND. What is left is let through to the screen.

For example: Color mask 11 = 1 + 2 + 8 (powers of 2) allows the following device colors to pass: 0, 1, 2, 3 (1+2), 8, 9 (1+8), 10 (2+8) and 11 (1+2+8) or color planes 0, 1 and 3.

The table below shows which color planes that are used (with plane oriented devices) and which device colors that are allowed to pass through the mask for some values of the color mask.
Ref 9-6  

MG - Window Mode

<table>
<thead>
<tr>
<th>Color mask</th>
<th>Color planes affected</th>
<th>&quot;Masked&quot; device colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0,1</td>
<td>1,2,3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0,2</td>
<td>1,4,5</td>
</tr>
<tr>
<td>6</td>
<td>1,2</td>
<td>2,4,6</td>
</tr>
<tr>
<td>7</td>
<td>0,1,2</td>
<td>1,2,3,4,5,6,7</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0,3</td>
<td>1,8,9</td>
</tr>
<tr>
<td>10</td>
<td>1,3</td>
<td>2,8,10</td>
</tr>
<tr>
<td>11</td>
<td>0,1,3</td>
<td>1,2,3,8,9,10,11</td>
</tr>
<tr>
<td>12</td>
<td>2,3</td>
<td>4,8,12</td>
</tr>
<tr>
<td>13</td>
<td>0,2,3</td>
<td>1,4,5,8,9,12,13</td>
</tr>
<tr>
<td>14</td>
<td>1,2,3</td>
<td>2,4,6,8,10,12,14</td>
</tr>
<tr>
<td>15</td>
<td>0,1,2,3</td>
<td>1,2,3,4,5,6,7,8,9,10,11,12,13,14,15</td>
</tr>
</tbody>
</table>

**Drawing Mode**

An object that is to be drawn is combined with graphics previously drawn on the screen in accordance with the current *drawing mode*. The drawing modes can be seen as bitwise logical operations between the object and the screen graphics. The following drawing modes are defined in **Multi-Graphics**:

*replace, stamp, mix, complement, mask* and *erase*.

When a window is created, the current drawing mode is set to *replace*.

It is not mandatory for screens to support all drawing modes. For instance, the visible screens on printers (i.e. the paper) usually support the *replace* drawing mode only. The available drawing modes can be read by calling *GetDrawModes*. 

*MultiGraphics & Borland Pascal/Turbo Pascal*  
*DATABITEN*
Replace

The drawing mode **replace** allows objects to be drawn without regard to existing screen graphics. New objects *replace* the graphics on the screen.

<table>
<thead>
<tr>
<th>Object</th>
<th>Screen</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>all device colors</td>
<td>screen</td>
<td>object</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Ref 9-8  MG - Window Mode

Stamp

The drawing mode **stamp** allows drawn objects to replace existing screen graphics except where the objects have a device color of 0. Screen graphics is not affected in these areas. The effect of this is that the new objects appear to lie in front of the screen background, i.e. they are "stamped" on the screen.

This drawing mode is useful when you want to show opaque icons, symbols and text that lie on top of the screen graphics. You can also make objects with "holes".

<table>
<thead>
<tr>
<th>Object device color&gt; 0</th>
<th>Screen screen</th>
<th>Result object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object device color= 0</th>
<th>Screen screen</th>
<th>Result screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object all device colors</th>
<th>Screen screen</th>
<th>Result object OR screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Complement

The drawing mode **complement** complements drawn objects with existing screen graphics using the bitwise logical operation **XOR**.

<table>
<thead>
<tr>
<th>Object</th>
<th>Screen</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>all device colors</td>
<td>screen</td>
<td>object XOR screen</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**complement** is very useful for producing animated graphics. If an object is drawn twice, the original screen graphics is restored. The technique is often used for cursors, "rubber band lines", icons etc. One disadvantage, however, is that the object and the screen graphics are mixed, which means the color of the drawn object is affected by the screen's background color.

Mask

The drawing mode **mask** "masks off" from the existing screen graphics the color information of objects using the bitwise logical operation **AND**. Those parts of objects that have a device color of 0 erase the screen, while parts with max. device color protect the screen from being affected etc.

The drawing mode can be used to protect the screen background, clip graphics against arbitrary geometric shapes etc..

<table>
<thead>
<tr>
<th>Object</th>
<th>Screen</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>all device colors</td>
<td>screen</td>
<td>object AND screen</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Erase

The drawing mode **erase** erases the existing screen graphics with the help of the logical bitwise operation **NOT** object **AND**. Those parts of the objects that have max. device color erase the screen, while parts with device color 0 protect the screen against being affected and so on.
The drawing mode can be used to erase parts of the screen background and protect others, clip graphics against arbitrary geometric shapes etc.

<table>
<thead>
<tr>
<th>Object</th>
<th>Screen</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>all device colors</td>
<td>screen</td>
<td>NOT object AND screen</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Constants**

- `minDrawMode` = `replace`
- `maxDrawMode` = `erase`

**Data Types**

```
DrawMode = (replace, stamp, mix, complement, mask, erase);
DrawModeSet = SET OF DrawMode;
```

Enumerated type of the drawing modes in **MultiGraphics** and the set type for available drawing modes.
Procedures & Functions

Current Position

**SetPos**

P SetPos (x, y: INTEGER);

This procedure sets the current drawing position in the current window. x and y are given as window coordinates.

When a window is created, the current position is set to (0,0) (this can be modified by automatically linked window capabilities).

The current position can be outside the window.

SV.Result
processWithoutWin the calling process has no window

**SetRelPos**

P SetRelPos (dx, dy: INTEGER);

This procedure sets the new current drawing position in the current window relative to the previous drawing position. dx and dy give the displacement in relation to the previous drawing position.

SV.Result
processWithoutWin the calling process has no window

MG.SetRelPos (-10, 5);
**MovePos**

```pascal
PROCEDURE MovePos(dist: INTEGER);
```

This procedure moves the current drawing position in the current window the distance `dist` from the previous drawing position. The displacement takes place in the current direction (set using `SetDir` or `TurnDir`).

The distance is measured in x units, which is important to take into consideration if the device's aspect ratio is different from 1:1 (1000).

**GetPos**

```pascal
PROCEDURE GetPos(x, y: INTEGER (*O*));
```

This procedure returns the current position in the current window. The position is given as window coordinates.

**CurX**

```pascal
FUNCTION CurX: INTEGER;
```

This function returns the current x position in the current window. The position is given as window coordinates.
The `CurY` function returns the current y position in the current window. The position is given as window coordinates.

```pascal
F CurY: INTEGER;
```

When an error occurs, `MAXINT` is returned.

The `GetFastXY` procedure returns proposals for nearby x and y window coordinates, based on specified x and y values, that can produce a higher drawing speed for drawing operations using patterns, bit mapped symbols, fonts, copying and images.

The x and y values returned are always <= the given values.

Many bit oriented graphics adapters such as CGA, Hercules, EGA and VGA work considerably faster if objects are drawn using byte boundaries for the x value. For these adapters, `GetFastXY` returns an x value, which is obtained from the following calculation 8*(x DIV 8). The y value is not changed.

When an error occurs, `MAXINT` is returned.

```
GetFastXY(VAR x,y: INTEGER (*IO*));
```

Moreover, errors that are specific for the device can be returned. When an error occurs, the original x and y are returned.

```
VAR
  x,y: INTEGER;
```
x := 100; y := 50;
MG.GetFastXY(x, y);
MG.SetPos(x, y);
MGImg.DrawImage(img);

Current Drawing Direction

<table>
<thead>
<tr>
<th>SetDir</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>P SetDir(angle: INTEGER);</td>
<td></td>
</tr>
</tbody>
</table>

This procedure sets the current drawing direction in the current window to the angle \( \text{angle} \), specified in tenths of a degree (decidegrees). The angle is normalized within the range of 0..3599 decidegrees.

The drawing direction affects among other things the drawing of lines using DrawLine, as well as text output.

```
SV.Result
processWithoutWin   the calling process has no window
MG.SetDir(450);
```

<table>
<thead>
<tr>
<th>TurnDir</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>P TurnDir(angle: INTEGER);</td>
<td></td>
</tr>
</tbody>
</table>

This procedure sets the current drawing direction in the current window by turning the current direction angle, \( \text{angle} \), specified in tenths of a degree (decidegrees). A positive turning angle changes the direction anti-clockwise, while a negative turning angle changes it in a clockwise direction.

The new angle is normalized within the range of 0..3599 decidegrees.

The drawing direction affects among other things the drawing of lines using DrawLine, as well as the printing of text.

```
SV.Result
processWithoutWin   the calling process has no window
MG.TurnDir(-10);
```
CurDir

F CurDir: INTEGER;

This function returns the current drawing direction as an angle measured in tenths of a degree (decidegrees). The angle lies within the range of 0..3599 decidegrees.

SV.Result
processWithoutWin the calling process has no window

Current Viewport

SetUpView

P SetUpView(x0,y0, width, height: INTEGER);

This procedure sets the desired viewport in the current window. All drawing operations are clipped against the current viewport. \((x0,y0)\) specifies the position for the lower left corner measured as window coordinates while \(width\) and \(height\) give the width and height respectively.

The viewport must lie within the current window's boundaries.

When a window is created, the viewport is automatically set to the same size as the window.

SV.Result
processWithoutWin the calling process has no window
invalidViewSpec invalid data for the viewport

Moreover, specific errors for each respective device are reported.

MG.SetView(50,50,100,100);
GetView

\[ P \text{ GetView} (\text{VAR } x_0, y_0, width, height: \text{ INTEGER } (*O*)) \];

This procedure returns the position and size for the current viewport in the current screen. \( x_0 \) and \( y_0 \) specify the viewport's lower left corner measured as window coordinates.

SV.Result

processWithoutWin the calling process has no window

VAR
  oldX0, oldY0 : INTEGER;
  oldWidth, oldHeight : INTEGER;
  ..
  MG.GetView(oldX0, oldY0, oldWidth, oldHeight);
  MG.SetView(...);
  ..
  MG.SetView(oldX0, oldY0, oldWidth, oldHeight);

Current Color Mask

SetColorMask

\[ P \text{ SetColorMask} (\text{colMask}: \text{ DeviceColor}); \]

This procedure sets the color mask in the current window.

The color mask decides which device colors in the foreground and background colors that are to be drawn. When using color plane oriented devices, the color mask will decide which color planes that are to be modified when drawing operations are performed.

The color mask is often used to protect color planes against being modified. Another reason for using this is to increase the speed of drawing when using color plane oriented screens.

When a window is created, the color mask = max. color mask is set for the screen. This is normally = max. device color. The max. color mask can be read using the function CurMaxColorMask (see "MG - Screens"). The default value means that all colors will be drawn (and that all color planes will be modified).
SV.Result
processWithoutWin the calling process has no window
invalidColorMask invalid color mask

MG.SetColorMask(1);
MGImg.DrawImage(img);

**CurColorMask**

F CurColorMask: DeviceColor;

This function returns the current color mask in the current window.

SV.Result
processWithoutWin the calling process has no window

When errors occur maxDeviceColor is returned.

**Current Drawing Mode**

**SetDrawMode**

P SetDrawMode(mode: DrawMode);

This procedure sets the drawing mode in the current window. Permitted
drawing modes for the current screen can be read using the procedure Get-
DrawModes.

When a window is created, replace is set as the default drawing mode.

The drawing mode affects all drawing operations.

SV.Result
processWithoutWin the calling process has no window
invalidDrawMode invalid drawing mode

MG.SetDrawMode(MG.complement);
MG.SetPos(50,50);
MG.DrawLineTo(100,50);
..
MG.SetPos(50,50);
MG.DrawLine(100,50);
CurDrawMode

F CurDrawMode: DrawMode;

This function returns the current drawing mode for the current window.

SV.Result
processWithoutWin the calling process has no window

When errors occur, maxDrawMode is returned.

GetDrawModes

P GetDrawModes(VAR modes: DrawModeSet (*O*));

After the call, the modes set variable contains the drawing modes permitted for the current window and screen.

SV.Result
processWithoutWin the calling process has no window

When an error occurs, modes is assigned the empty set.

VAR
drwModes: MG.DrawModeSet;
...
MG.GetDrawModes(drwModes);
IF MG.stamp IN drwModes THEN
    MG.SetDrawMode(MG.stamp);..
The most elementary drawing operation is the drawing of a single point. On raster devices, to draw a point means to set the color of a pixel ("picture element").

As the resolutions of graphics devices increases, however, simple point drawing is being used less frequently, simply because the points are becoming too small.

Points, in MultiGraphics, are drawn at a specified position and with a specified device color (DrawPoint), or at the current position in the current window using the current foreground color and drawing mode (DrawCurPoint).

In MultiGraphics, there is also support for reading the device color of a point (pixel for raster oriented devices).

Finally, MultiGraphics also supports outline tracing (TraceOutline and TraceComplexOutline) - an unique technique for tracing the contours of bitmapped screen objects and returning their coordinates as a point array. Later, this array may be processed using mathematical tools (transforms, polynomial approximations, Bézier approximations with MGBez, smoothing, etc.).
Data Types

Point = RECORD x,y: INTEGER END;
ShortPoint = RECORD x,y: ShortInt END;

These are data types used when defining polylines, polygons and bezier elements (see "MG - Lines"). In MultiGraphics, points are constructed of both integer coordinates and short integer coordinates.

OnOutlineProc = FUNCTION(pointCol: DeviceColor;
                          x       : INTEGER;
                          y       : INTEGER):
                          BOOLEAN;

Data type to declare your own functions to check if points belong to the outline or not, when calling TraceComplexOutline. When called, pointCol contains the device color of the checked point, while x and y contain the window coordinates of the point. The function shall return TRUE if the point belongs to the outline - otherwise FALSE.

Constants

nullPoint      : Point = (x:-32768; y:-32768);
nullShortPoint : ShortPoint = (x:-128; y:-128);

The constants are used to denote points to be skipped in polyline and polygon drawing (see DrawPolyline, DrawShortPolyline, FillPolygon and FillShortPolygon).

NullOnOutline;

Constant to specify a non-existant outline check function when calling TraceComplexOutline. The default check in TraceOutline is used in it's place.
Procedures & Functions

**DrawCurPoint**

```pascal
P DrawCurPoint;
```

This procedure draws a point at the current position in the current window, using the current foreground color, color mask and drawing mode. If the current position lies outside the current viewport, nothing is drawn.

```pascal
SV.Result
processWithoutWin       the calling process has no window
deviceProcNotInUse     device's driver does not support this procedure
```

Errors that are specific for the current device are also reported.

```pascal
MG.SetPos(100,100);
MG.MatchForeground(MG.red);
MG.SetDrawMode(MG.replace);
MG.DrawCurPoint;
```

**DrawPoint**

```pascal
P DrawPoint (x,y : INTEGER;
   color: DeviceColor);
```

This procedure draws a point at window position (x,y) in the current window, with the color `color`, using the current color mask and drawing mode. If the position lies outside the current viewport, nothing is drawn.

```pascal
The current position and device color are not modified!
```

```pascal
SV.Result
processWithoutWin       the calling process has no window
invalidDeviceColor     device color not defined
deviceProcNotInUse     device's driver does not support this procedure
```

Errors that are specific for the current device are also reported.

```pascal
MG.SetDrawMode(MG.replace);
MG.DrawPoint(100,100,14);
```
CurPointDeviceColor

F CurPointDeviceColor: DeviceColor;

This function returns the device color for the point (pixel) at the current position in the current window. The current position must lie within the current viewport.

The function is not implemented for all device types and screens (the visible screen of printers, plotters, etc.).

SV.Result
processWithoutWin the calling process has no window
deviceProcNotInUse device’s driver does not support this procedure
outsideView the current position lies outside the viewport

Errors that are specific for the current device are also reported. When an error occurs, the value maxDeviceColor is returned.

MG.SetForeground(8);
IF MG.CurPointDeviceColor = 15 THEN
  MG.DrawCurPoint;

PointDeviceColor

F PointDeviceColor(x,y: INTEGER): DeviceColor;

This function returns the device color for a point (pixel) at position (x,y) in the current window. The specified position must lie within the current viewport.

The function is not implemented for all device types and screens (the visible screen of printers, plotters, etc.).
Errors that are specific for the current device are also reported. When an error occurs, the value \texttt{maxDeviceColor} is returned.

\begin{verbatim}
IF MG.PointDeviceColor(100,50) = 15 THEN
  MG.DrawPoint(100,50,8);
\end{verbatim}

Outline Tracing

\begin{verbatim}
P TraceOutline(   x0,y0    : INTEGER;
                 firstIdx: Word;
                 maxN    : Word
                 VAR outline ;         (*O*)
                 VAR n       : Word     (*O*)
)
\end{verbatim}

This procedure traces and returns the contour points which, taken as a whole, make up the outline of an object in the current window. The position of the first contour point is specified in \((x0,y0)\) which has to reside inside the current viewport. The procedure searches for neighboring points having the same device color as the starting point. The tracing is terminated when the outline becomes closed, i.e. when \texttt{TraceOutline} has returned to the starting point.

The window coordinates of the outline points are written to the variable parameter \texttt{outline} which should define an array of points (\texttt{MG.Point}) with start index = 0 so dimensioned that the array may contain all traced points. The index of the first outline point is specified with \texttt{firstIdx} ( \(>= 0\)). The maximum index allowed is specified in \texttt{maxN}. Usually \texttt{maxN} = the number of elements in the current \texttt{outline} parameter reduced by 1.

The number of outline points \texttt{traced} is returned in \texttt{n}.

Outline tracing can only be used with devices supporting the function \texttt{PointDeviceColor}. 
SV.Result
processWithoutWin the calling process has no window
invalidIndex invalid starting index (firstIndex)
outsideView starting point outside viewport
deviceProcNotInUse driver lacks support of outline tracing
invalidStartPoint invalid starting point (no contour found)

CONST
maxPoints = 500;
VAR
  outline: ARRAY[0..maxPoints - 1] OF MG.Point;
  n      : Word;
..
MG.TraceOutline(102,251,0,maxPoints - 1,outline,n);
IF SV.Result = SV.ok THEN BEGIN
  MG.SetPos(50,50);
  MG.FillPolygon(outline,0,n - 1);
 ..

TraceComplexOutline

P TraceComplexOutline( x0,y0 : INTEGER;
OnOutline: OnOutlineProc;
firstIdx : Word;
maxN     : Word
VAR outline; (*O*)
VAR n        : Word    (*O*));

Same as TraceOutline, but you can specify the conditions on when a point
belongs to the outline with a user-defined, FAR-declared, function
OnOutline of type OnOutlineProc. This makes it possible to adapt the
outline tracing to your own needs. You can, for instance, trace raster images
where the outline is specified as points belonging to a specific intensity
interval, a specific color interval, etc.

SV.Result
processWithoutWin the calling process has no window
invalidIndex invalid starting index (firstIndex)
outsideView starting point outside viewport
deviceProcNotInUse driver lacks support of outline tracing
invalidStartPoint invalid starting point (no contour found)
FUNCTION IsGrayOrWhite(pointCol: MG.DeviceColor; 
    x,y : INTEGER): BOOLEAN;
BEGIN
(* Default VGA palette *)
  IsGrayOrWhite:= Word(pointCol) IN [7,8,15];
END;

VAR
  x0,y0 : INTEGER;
  outline: ARRAY[0..1000] OF MG.Point;
  n     : Word;
...
MG.TraceComplexOutline(x0,y0, 
  IsGrayOrWhite,  
  0, 
  1000, outline,  
  n);

See GAOUTLIN.PAS for more examples.
MultiGraphics contains several line based drawing procedures. Among these are the following procedures for drawing:

- straight lines,
- polylines,
- Bézier curves (polybéziers),
- rectangles (boxes),
- circles,
- arcs,
- sectors of circles,
- ellipses,
- elliptical arcs and
- sectors of ellipses.

All line graphics is clipped against the current viewport and drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode as well as the special attributes for line styles and line widths.

When a window is created, the line style is set to solid and the line width to 1 unit (1 pixel).

Both start and end positions can lie outside the current viewport and window.
Bézier Curves

Bézier curves have many nice properties that make them a good choice for description of and modelling geometrical shapes in general, and especially for symbol and font description.

The Bézier curves we will use here are defined as cubic polynomials, i.e. polynomials of the third degree. Mathematically speaking, these curves can be defined by specifying the coordinates of only four points, the so called control points. The array type Bezier, defined in MG, is an array of four points, which you can use to declare variables and parameters for Bézier curves.

The first control point is at the start point of the Bézier curve, while the last control point (the fourth) is at the endpoint of the curve. The two middle control points work as "magnets" that exert a "pull" on the curve. Expressed in a more stringent style, the tangent of the curve at the first control point always passes through the second control point, while the tangent of the curve at the last control point pass through the third control point.

The example program GABez lets you interactively modify a Bézier curve by moving the control points with the mouse.
We call an array of Bézier curves for a *polybézier curve*. The elements of this array are called *Bézier elements*. A polybézier is continuous when the endpoint of one Bézier element, at the same time, is the start point of the next element, otherwise the polybézier is discontinues. If you make a proper selection of control points the tangent directions of the two linked curve element may coincide, with a resulting smooth transition from one curve to another (see fig. 2).

![Fig 2: Two Bézier curves with the same tangent directions at their common point](image-url)

Polybézier curves are drawn using `DrawPolyBezier`.

**Examples**

`GALine` gives an example of line and circle drawing with line patterns. `GACircle` displays circles and ellipses. `GAPoly` draws polylines. `GABez` is an example of interactive drawing of Bézier curves - recommended if you want a better understanding of Bézier curves. `GAPolyBe` draws and fills polybézier curves.
Data Types

LineStyle;

This is the abstract data type for line styles. The actual implementation depends on the type of device (raster, vector etc.). See "MGPat - Pattern" and interface section for details.

The data types Point and ShortPoint that are used to specify polylines, are defined in "MG - Points".

Constants

nullLineStyle

Specifies the absence of a line style.

Procedures & Functions

Line Properties

Line Style

SetLineStyle DLR

P SetLineStyle(style: LineStyle);

Sets the current line style in the current window. The line style is specified using a parameter of the pointer type LineStyle. The position in the line pattern is set to the start, i.e. 0.

When a window is created, the current line style is set to solid.

Four generic, i.e., device independent, line styles are defined in the unit
**MG - Lines**

**MGPat.** These are **solid**, **dotted**, **dashed** and **dashedDotted** (see "MGPat - Pattern"). Generic line styles can be set directly using **MGPat.SetGenLineStyle**.

You can create your own line styles if you have information about the line style structure of the specific device and screen. See **MGPat** and the description of each respective driver for more details.

**SV.Result**  
**processWithoutWin** the calling process has no window

```pascal
MG.SetLineStyle(MGPat.GenLineStyle(MGPat.dotted));
```

or

```pascal
MGPat.SetGenLineStyle(MGPat.dotted);
```

**CurLineStyle**

```pascal
F CurLineStyle:LineStyle;
```

Returns a reference to the line style in the current window.

**SV.Result**  
**processWithoutWin** the calling process has no window

If an error occurs, **nullLineStyle** is returned.

**SetLineStylePos**

```pascal
P SetLineStylePos(startPos:Word);
```

Sets the position in the current line style pattern. When you first select a line pattern, the position is set to 0. The position is updated during line drawing, if the line style protection mode is switched off.

**SV.Result**  
**processWithoutWin** the calling process has no window  
**invalidLineStylePos** invalid position in line pattern
Ref 11-6

MGPat.SetGenLineStyle(MGPat.dashedDotted);
MG.SetLineStylePos(3);
MG.DrawLineTo(50,75);

CurLineStylePos

F CurLineStylePos: Word;

Returns the current position in the line style pattern of the current window.

SV.Result
processWithoutWin the calling process has no window

If an error occurs, 65535 is returned.

VAR
  stylePos: Word;
  ..
  stylePos:= MG.CurLineStylePos;
  MG.DrawCircle(100);
  MG.SetLineStylePos(stylePos);

SetProtectLineStyle

P SetProtectLineStyle(onoff: BOOLEAN);

Sets the line style protection mode on (onoff has the value TRUE or on) or switches it off (onoff has the value FALSE or off).

When a window is created, the line style protection mode is switched off (FALSE) by default.

With line style protection mode on, each new line will start from the same position in the line style pattern.

With line style protection mode off, each new line will continue from the position in the line style pattern where the previous line stopped. This is very useful when drawing continuous lines in polylines, polygons etc..

SV.Result
processWithoutWin the calling process has no window
CurProtectLineStyle

F CurProtectLineStyle: BOOLEAN;

Returns the setting for the line style protection mode in the current window. If FALSE, the protection mode is switched off. If TRUE, the protection mode is switched on.

SV.Result
processWithoutWin the calling process has no window

If an error occurs, FALSE is returned.

SetEndPointSkip

P SetEndPointSkip(trueOrFalse BOOLEAN);

Decides if the line drawing procedures in the current window shall skip the line endpoints (trueOrFalse = TRUE) or not (trueOrFalse = FALSE). Influences DrawLineTo, DrawLineRel and DrawLine.

When a window is created, line skip is set to off (FALSE) by default.

The endpoint of a line is only skipped if the point is visible in the current viewport.

This procedure is useful when you intend to draw a series of linked lines. The endpoint of one line is at the same time the starting point of the next line and would thereby be drawn twice if not skipped. This would shift the line style pattern one step. Also, when using the drawing mode complement, the starting point would in fact erase the endpoint leaving a visible hole between the lines.

SV.Result
processWithoutWin the calling process has no window

MG.SetEndPointSkip(TRUE);
MG.SetDrawMode(MG.complement);
MG.DrawLineRel(100,0);
MG.DrawLineRel(-50,100);
MG.DrawLineRel(-50,-100);
CurEndPointSkip

F CurEndPointSkip: BOOLEAN;

Returns the current endpoint skip mode for line drawing in the current window. If FALSE endpoints are not skipped, if TRUE they are skipped.

SV.Result
processWithoutWin the calling process has no window

If an error occurs, FALSE is returned.

Line Width

SetLineWidth

P SetLineWidth(w: INTEGER);

Sets the line width in the current window. The line width must be >= 1.

For raster devices, the line width is specified in pixels. For other devices, the unit for measuring width is device dependent.

When a window is created, the line width is assigned to 1 unit.

SV.Result
processWithoutWin the calling process has no window
invalidLineWidth invalid line width

MG.SetLineWidth(3);
MG.DrawLine(50);

CurLineWidth

F CurLineWidth: INTEGER;

Returns the line width in the current window.

SV.Result
processWithoutWin the calling process has no window

If an error occurs, MAXINT is returned.
Straight Lines

**DrawLineTo**

\[ \text{P } \text{DrawLineTo}(x, y: \text{INTEGER}); \]

Draws a straight line from the current position to \((x,y)\). The position is given in window coordinates. The line is clipped against the current viewport and drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style position, line style protection, line width, and endpoint skip mode.

The new current position is \((x,y)\), even if this endpoint lies outside the viewport or window.

If line style protection mode is switched off, the current line pattern position is modified. If the line style protection mode is on, the original line pattern position is retained.

The endpoint is skipped if endpoint skip mode is on (TRUE).

**SV.Result**

**process WithoutWin**  
the calling process has no window

Errors that are specific for the device are also returned.

```pascal
MG.SetPos(100,100);
MG.DrawLineTo(150,100); MG.DrawLineTo(150,100);
MG.DrawLineTo(100,150); MG.DrawLineTo(100,100);
```

**DrawLineRel**

\[ \text{P } \text{DrawLineRel}(dx, dy: \text{INTEGER}); \]

Draws a straight line *relative* from current position \((x0,y0)\) to \((x0 + dx, y0 + dy)\). The displacement is given in window units.

The line is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style position, line style protection, line width, and endpoint skip mode.

The new current position is \((x0 + dx,y0 + dy)\), even if this endpoint lies
outside the viewport or window.

If line style protection mode is switched off, the current line pattern position is modified. If the line style protection mode is on, the original line pattern position is retained.

The endpoint is skipped if endpoint skip mode is on (TRUE).

```
SV.Result
processWithoutWin the calling process has no window
```

Errors that are specific for the device are also returned.

```
MG.SetPos(100,100);
MG.DrawLineRel(50,0); MG.DrawLineRel(0,50);
MG.DrawLineRel(-50,0); MG.DrawLineRel(0,-50);
```

<table>
<thead>
<tr>
<th>DrawLine</th>
<th>DL</th>
</tr>
</thead>
</table>

```
P DrawLine(dist: INTEGER);
```

Draws a straight line of the length `dist` from the current position and in the current drawing direction. The distance is given in x units.

The line is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style position, line style protection, line width, and endpoint skip mode.

The new current position becomes the line's endpoint even if this endpoint lies outside the viewport or window.

If line style protection mode is switched off, the current line pattern position is modified. If the line style protection mode is on, the original line pattern position is retained.

The endpoint is skipped if endpoint skip mode is on (TRUE).
SV.Result
processWithoutWin the calling process has no window

Errors that are specific for the device are also returned.

MG.SetPos(100,100);
MG.SetDir(0);   MG.DrawLine(50);
MG.SetDir(900); MG.DrawLine(50);
MG.SetDir(1800); MG.DrawLine(50);
MG.SetDir(2700); MG.DrawLine(50);

Polylines

P TransformPolyline(VAR polyline;{ARRAY OF MG.Point} (*IO*)
  first,last : Word;
  scaleX,scaleY: Integer32;
  angle        : INTEGER;
  shearX,shearY: INTEGER);

Transforms the points with index first to last (last >= first) in the point array polyline. The points are scaled in the x direction by the factor scaleX (1000 = unscaled) and in the y direction by scaleY (1000 = unscaled). The points are rotated by angle deci degrees (clockwise). The points are sheared in the x direction by shearX (0 = unsheared) and in the y direction by shearY (0 = unsheared).

A special point, MG.nullPoint, is used to separate different polylines in the array of points from each other. MG.nullPoint is not transformed.

SV.Result
MGErr.invalidIndex         last >= first!
MGErr.invalidAngle         0<= angle < 3600
MGErr.invalidTransform     integer overflow

pa: ARRAY[0..999] OF MG.Point;
..
MG.TransformPolyline(pa,0,299,2000,2000,900,0,0);
MG.DrawPolyline(pa,0,299);
DrawPolyline

P DrawPolyline(VAR polyline; (*I*)
    first, last: Word);

Draws a polyline relative the current position and the array of relative points found in the variable parameter polyline. The polyline starts from the point with the index first and ends at the point with the index last, where last >= first. The polyline is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style position, line style protection, line width, and endpoint skip mode.

The new current position is the point (relative to the current start position) which is given in the final index last. The line pattern is up-dated if the line style protection mode has not been set to on.

polyline must be of the type ARRAY [0..maxindex] OF Point, i.e. an array with start index 0 and a maximum index, called maxindex, which is dependent on the number of points in the array. The elements of the array must be of the type Point (see "MG - Points" under "Data Types").

The coordinates of the points do not specify absolute coordinates, instead these are given relative to the current window position.

Suppose we have declared an array of points as follows:

((0,5), (10,0), (0,15), (-10,0), (10,0), (-32768,-32768), (0,5), (0,10))

and the current position is (100,100). If the start index = 1 and final index = 4, the poly-lines will then connect the following points:

(110,100) - (100,115) - (90,100) - (110,100)

One particular point, called nullPoint ((-32768,-32768), see "MG - Points" under "Constants"), is treated differently so that the polyline make a break at this point and start afresh at the next point. nullPoint is used to draw several separate polylines using one single call by DrawPolyline. This is very useful when drawing symbols and characters.
Using the same array of points as above, but with a new final index = 7, the following points are connected:

(110,100) – (100,115) – (90,100) – (110,100) and
(100,105) – (100,110)

If last < first nothing happens. No error message is given.

Integer overflow in the addition of point data with the current position is not reported!

SV.Result
processWithoutWin the calling process has no window

Errors are also reported that are specific for each respective driver.

TYPE
  PolyArray = ARRAY[0..11] OF MG.Point;
CONST
CONST
  minInt = -32768;
  pa : PolyArray = (  
    (x: 0;y: 0),
    (x:30;y: 0),
    (x:30;y:30),
    (x: 0;y:30),
    (x: 0;y: 0),
    (x:minInt;y:minInt),
    (x:10;y:10),
    (x:20;y:10),
    (x:20;y:20),
    (x:10;y:20),
    (x:10;y:10),
    (x:minInt;y: minInt));
 ..
MGPat.SetGenPattern(MGPat.vert);
MG.SetPos(100,200);
MG.DrawPolyline(pa,0,11); (* Small and large square *)
MG.SetPos(200,200);
MG.DrawPolyline(pa,0,5);  (* Large square *)
MG.SetPos(300,200);
MG.DrawPolyline(pa,5,11); (* Small square *)
TransformShortPolyline

```pascal
P TransformShortPolyline(
    VAR shortPoly; (* ARRAY OF MG.ShortPoint*) (*IO*)
    first, last : Word;
    scaleX, scaleY: Integer32;
    angle : INTEGER;
    shearX, shearY: INTEGER);
```

Transforms a "short" point array `shortPoly`.

A special point, `MG.shortNullPoint`, is used to separate different polylines in the array of points from each other. `MG.shortNullPoint` is not transformed.

See TransformPolyline.

DrawShortPolyline

```pascal
P DrawShortPolyline(VAR shortPoly; (*I*)
    first, last: Word);
```

Draws a polyline based on the current position and the array of relative "short" points found in the variable parameter `shortPoly`. The polyline starts from the point with the index `first` and ends at the point with the index `last`, where `last >= first`. The polyline is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style position, line style protection, line width, and endpoint skip mode.

The new current position becomes the point (relative to the current start position) that is specified by the final index `last`. The line pattern is updated if the line style protection mode has not been set to on.

`shortPoly` must be of the type `ARRAY [0..maxindex] OF ShortPoint`, i.e. an array with a start index 0 and a maximum index, called `maxindex`, which is dependent on the number points in the array. The elements of the array must be of the type `ShortPoint` (see "MG - Points" under "Data Types").

The coordinates of the points do not specify absolute coordinates, instead these are given relative to the current window position.

Suppose we have declared an array of points as follows:
and the current position is (100,100). If the start index = 1 and the final index = 4, the polylines will connect the following points:

(110,100) - (100,115) - (90,100) - (110,100)

One particular point, called nullShortPoint ((-128,-128), see "MG - Points" under "Constants"), is treated differently so that the polyline make a break at this point and start afresh at the next point. nullShortPoint is used to draw several separate polylines using one single call by DrawShortPolyline. This is very useful when drawing symbols and characters.

Using the same array of points as above, but with a new final index = 7, the following points are connected:

(110,100) - (100,115) - (90,100) - (110,100) and (100,105) - (100,110)

If last < first nothing happens. No error message is given.

Integer overflow in the addition of point data with the current position is not reported!

SV.Result
processWithoutWin the calling process has no window

Errors are also reported that are specific for each respective driver.

Rectangle

\[
\text{DrawBox}(\text{width}, \text{height} : \text{INTEGER});
\]

Draws a rectangle (box) using the sizes width and height respectively, with the starting corner in the current window position and the base in the x direction. The width and height are measured in x units and y units respectively.
The box is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style position, line style protection and line width.

The current position and direction are not modified.

The line pattern is updated if the line style protection mode has not been set to on.

*The base is drawn in the \( x \) direction. The height is measured in \( y \) units.*

**SV.Result**

`processWithoutWin` the calling process has no window

Errors are also reported that are specific for each respective driver.

```pascal
MG.SetPos(100,100);
MG.SetLineWidth(2);
MG.DrawBox(100,50);
```

**DrawRect**

```pascal
P DrawRect(width, height: INTEGER);
```

Draws a rectangle using the sizes `width` and `height` respectively, with the starting corner in the current window position and the base in the *current drawing direction*. The lengths of the sides are measured in *\( x \) units*.

The box is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style protection and line width.

The current position and direction are not modified.

The line pattern is updated if the line style protection mode has not been set to on.

*The base is drawn in the current direction. The height is measured in \( x \) units.*

**SV.Result**

`processWithoutWin` the calling process has no window

Errors are also reported that are specific for each respective driver.
Circles, Arcs and Sectors

**DrawCircle**

```pascal
P DrawCircle(radius: INTEGER);
```

Draws a *circle* using the radius `radius` and with its center at the current position. The radius is specified in x units.

The circle is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style and line width.

The current position and direction are not modified.

The line pattern is up-dated if the line style protection mode has not been set to on.

**SV.Result**

- **processWithoutWin**: the calling process has no window
- **invalidRadiusOrAxis**: the radius must be > 0

Errors are also reported that are specific for each respective driver.

```pascal
MG.SetPos(200,200);
MGPat.SetGenLineStyle(MGPat.dashed);
MG.DrawCircle(100);
```

**DrawArc**

```pascal
P DrawArc(radius, centralAngle: INTEGER);
```

Draws a circular *arc* with its center at the current window position, the starting angle is the same as the current drawing direction, the radius `radius` and with a central angle, `centralAngle`, measured in tenths of degrees (decidegrees). The radius is given in x units.
A positive central angle draws the arc anti-clockwise, while a negative angle draws it clockwise.

The arc is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style and line width.

The current direction is set to the arc's final direction. The direction is normalized within the range of 0..3599 decidegrees. The current position is not modified however.

The line pattern is up-dated if the line style protection mode has not been set to on.

SV.Result
processWithoutWin the calling process has no window
invalidRadiusOrAxis the radius must be > 0

Errors are also reported that are specific for each respective driver.

MG.SetPos(200,200);
MG.MatchForeground(MG.green);
MG.SetDir(1350);
MG.DrawArc(100,-450);

Fig 3: Arc and sector
**MG - Lines**

**DrawSector**

\[ P \text{ DrawSector}(\text{radius, centralAngle}: \text{INTEGER}); \]

Draws a *circle sector* with its center at the current window position, the starting angle the same as the current drawing direction, the radius `radius` and with a central angle, `centralAngle`, measured in tenths of degrees (decidegrees). The radius is given in x units.

A positive central angle draws the sector anti-clockwise, while a negative angle draws it clockwise.

The sector is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style and line width.

The current direction is set to the final direction of the sector's arc. The direction is normalized within the range of 0..3599 decidegrees. The current position is not modified however.

The line pattern is up-dated if the line style protection mode has not been set to on.

**SV.Result**

- `processWithoutWin` the calling process has no window
- `invalidRadiusOrAxis` the radius must be > 0

Errors are also reported that are specific for each respective driver.

MG.SetPos(200,200);
MG.MatchForeground(MG.green);
MG.SetDir(1350);
MG.DrawSector(100,-450);

**Ellipses, Elliptical Arcs and Sectors of Ellipses**

**DrawEllipse**

\[ P \text{ DrawEllipse}(a,b,\text{majorAxisAngle}: \text{INTEGER}); \]

Draws an *ellipse* with the *major axis* `a` and the *minor axis* `b`, with the direction of the major axis in accordance with `majorAxisAngle` in tenths of
degrees (decidegrees) and with its center at the current position. The major and minor axes are specified in x units.

![Ellipse Diagram](image)

**Fig 4: Ellipse**

The ellipse is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style and line width.

The current position and direction are not modified.

The line pattern is up-dated if the line style protection mode has not been set to on.

**SV.Result**

- `processWithoutWin` the calling process has no window
- `invalidRadiusOrAxis` the major and minor axes must be > 0

Errors are also reported that are specific for each respective driver.

```pascal
MG.SetPos(200,200);
MG.SetLineWidth(3);
MG.DrawEllipse(100,50,450);
```
DrawEllipseArc

\textbf{P DrawEllipseArc}(a, b, \text{majorAxisAngle}: \text{INTEGER}; \\
\text{centralAngle} \quad : \quad \text{INTEGER});

Draws an \textit{elliptical arc} with its center at the current window position, the starting angle the same as the current drawing direction, the major axis \textit{a}, the minor axis \textit{b}, the major axis direction \textit{majorAxisAngle} and with a central angle, \textit{centralAngle}, measured in tenths of degrees (decidegrees). The major and minor axes are specified in x units.

A positive central angle draws the sector anti-clockwise, while a negative angle draws it clockwise.

The elliptical arc is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style and line width.

The current direction is set to the arc's final direction. The direction is normalized within the range of 0..3599 decidegrees. The current position is not modified however.

The line pattern is up-dated if the line style protection mode has not been set to on.

\textbf{SV.Result}
\textbf{processWithoutWin} the calling process has no window
\textbf{invalidRadiusOrAxis} the radius must be > 0

Errors are also reported that are specific for each respective driver.

DrawEllipseSector

\textbf{P DrawEllipseSector}(a, b, \text{majorAxisAngle}: \text{INTEGER}; \\
\text{centralAngle} \quad : \quad \text{INTEGER});

Draws a \textit{sector of an ellipse} with its center at the current window position, the starting angle the same as the current drawing direction, major axis \textit{a}, minor axis \textit{b}, the major axis direction \textit{majorAxisAngle} and with a central angle, \textit{centralAngle}, measured in tenths of degrees (decidegrees). The major and minor axes are specified in x units.

A positive central angle draws the sector anti-clockwise, while a negative
angle draws it clockwise.

The elliptical sector is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style and line width.

The current direction is set to the final direction of the sector's arc. The direction is normalized within the range of 0..3599 decregs. The current position is not modified however.

The line pattern is up-dated if the line style protection mode has not been set to on.

SV.Result
processWithoutWin the calling process has no window
invalidRadiusOrAxis the radius must be > 0

Errors are also reported that are specific for each respective driver.

MG.SetPos(200,200);
MG.MatchForeground(MG.green);
MG.SetDir(450);
MG.DrawEllipseSector(100,50,450);

Bézier Curves

**TransformPolyBezier**

```p
P TransformPolyBezier(VAR polyline;{ARRAY OF MG.Bezier})
  first,last : Word;
  scaleX, scaleY: Integer32;
  angle : INTEGER;
  shearX, shearY: INTEGER);
```

Transforms the Bézier elements with index first to last (last >= first) in the Bézier array polyBez. The Bézier points are scaled in the x direction by the factor scaleX (1000 = unscaled) and in the y direction by scaleY (1000 = unscaled). The points are rotated by angle deci degrees (counter clockwise). The points are sheared in the x direction by shearX (0 = unsheared) and in the y direction by shearY (0 = unsheared).
SV.Result
MGErr.invalidIndex       last >= first!
MGErr.invalidAngle       0<= angle < 3600
MGErr.invalidTransform   integer overflow

pb: ARRAY[0..499] OF MG.Bezier;
...
MG.TransformPolyBezier(pb,100,199,500,500,450,0,0);
MG.DrawPolyBezier(pa,100,199);

**DrawPolyBezier**

P DrawPolyBezier(VAR polyBez; (* ARRAY OF Beziers*) (*I*)
   first, last: Word);

Draws a polybézier curve relative the current position and the array of relative Bézier elements found in the variable parameter polyBez. The polybézier starts from the Bézier element with the index first and ends with the element with the index last, where last >= first. The polybézier is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style position, line style protection, line width, and endpoint skip mode.

The new current position is the point (relative to the current start position) which is given as the last control point of the Bézier element with index last. The line pattern is up-dated if the line style protection mode has not been set to on.

The only permitted start index of the polybézier array is 0. The coordinates of the control points do not specify absolute coordinates, instead these are given relative to the current window position.

SV.Result
processWithoutWin         the calling process has no window
tooManyPoints             internal buffer filled up

Errors are also reported that are specific for each respective driver.
TYPE
  PolyBez = ARRAY[0..4] OF MG.Bezier;
CONST
  pb: PolyBez = (((x:10;y:250),(x:50;y:400),
    (x:250;y:400),(x:300;y:250)),
    ((x:300;y:250),(x:250;y:50),
    (x:50;y:50),(x:10;y:250)));

  MG.SetPos(50,0);
  MG.DrawPolyBezier(pb,0,1);
MultiGraphics contains procedures for filling the following shapes with patterns:

- rectangles,
- polygons,
- Bézier curves (polybéziers),
- circles,
- sectors of circles,
- ellipses,
- sectors of ellipses and
- irregular shaped areas.

All pattern filling is clipped against the current viewport and is drawn using the current window mode, i.e. using the current foreground and background colors, color mask and drawing mode as well as the current pattern.

When a window is created, the pattern type is set to solid.

All patterns are monochrome. Bit oriented patterns are drawn using the current foreground color where the pattern is set to one, while those parts set to zero are drawn using the background color.

See also "MGPat - Standard Patterns".

Bézier Curves

For a description of Bézier curves, see "MG - Lines".
Examples

**GAPat** displays many of the standard patterns. **GAPoly** shows filling of polygons and **GAPolyBe** filling of polybéziers. **GACcharts** makes use of pie filling (pie charts).

Data types

**Pattern;**

This is the hidden data (pointer) type for patterns. The implementation depends on the type of device. See "MGPat - Standard patterns" and the interface section for **MGPat** for details.

Constants

**nullPattern**

Specifies a non-existant pattern. Often used as the current pattern to skip filling.

Procedures & Functions

The Current Pattern

**SetPattern**

```pascal
P SetPattern (pat: Pattern);
```

This procedure sets the current pattern in the current window. The pattern is specified using `pat`.

You are even allowed to set an empty pattern, i.e. switch off pattern filling.
This is done by giving `nullPattern` as a parameter.

When a window is created, the current pattern is set to `solid`.

In the unit `MGPat`, you will find a large number of generic patterns that are device independent. Generic patterns can also be set directly by calling `MGPat.SetGenPattern`. See "MGPat - Standard patterns". See also the specifications for each driver regarding their internal pattern structure.

```pascal
SV.Result
processWithoutWin  // the calling process has no window
    MG.SetPattern(MGPat.zigzag8);
    MG.SetPattern(MGPat.GenPattern(MGPat.cross));

or

    MGPat.SetGenPattern(MGPat.cross);
```

```pascal
CurPattern

F CurPattern: Pattern;

This function returns a reference to the current pattern in the current window.

SV.Result
processWithoutWin  // the calling process has no window

VAR
    oldPat: MG.Pattern;
    ..
    oldPat:= MG.CurPattern;
    MG.SetPattern(MGPat.GenPattern(MGPat.halftone5));
    ..
    MG.SetPattern(oldPat);
```
Filling Rectangles

**FillView**

```
P FillView;
```

This procedure fills the viewport in the current window with the current pattern. The pattern is drawn using the current window mode, i.e. using the foreground and background colors, color mask and drawing mode. No border line is drawn.

The current position is not changed.

**SV.Result**
- `processWithoutWin` the calling process has no window
- `deviceProcNotInUse` the driver does not support the procedure

Errors that are specific for each driver are also reported.

```pascal
MG.SetView(50, 50, 100, 75);
MGPat.SetGenPattern(MGPat.diagUp);
MG.MatchForeground(MG.cyan); MG.MatchBackground(MG.black);
MG.FillView;
```

**FillRect**

```
P FillRect(x0, y0, width, height: INTEGER);
```

This procedure fills the specified rectangular area in the current window with the current pattern. \((x0, y0)\) specifies the rectangle's lower left corner. \(width\) and \(height\) specify the rectangle's width and height respectively.

The rectangular area is clipped against the current viewport and is drawn using the current window mode, i.e. using the foreground and background colors, color mask and drawing mode. No border line is drawn.

The current position is not changed.

**SV.Result**
- `processWithoutWin` the calling process has no window
deviceProcNotInUse
deviceProcNotInUse the driver does not support the procedure

Errors that are specific for each driver are also reported.

**FillPolygon**

\[
P \text{FillPolygon}(\text{VAR vertices};(*I*)) \]

\[
\text{first, last: Word)}; 
\]

This procedure fills a *polygon* using the current position and the array of *relative* point data which is found in the variable parameter *vertices*. The polygon starts from the point with the index *first* and ends at the point with the index *last*, where *last* \( \geq \) *first*.

If the endpoint does not coincide with the starting point then the polylines are automatically joined to form a polygon.

The polygon is clipped against the current viewport and is filled using the current window mode, i.e. the foreground and background color, color mask and drawing mode. No border line is drawn.

The current position and drawing direction are not changed.

*vertices* must be of the type ARRAY [0..maxindex] OF Point, i.e. an array with a start index of 0 and a maximum index, called here the *maxindex*, which depends on the number points in the array. The number of points must be at least 3. The elements of the array must be of the type Point (see "MG - Points" under "Data types").

The coordinates of the points are not given as absolute coordinates, but are instead *relative* to the current window position.

Suppose that we have declared an array of points as follows:

\[
((0,5), (10,0), (0,15), (-10,0), (-32768,-32768), (0,5), (0,10), (-5,5))
\]

and the current position is (100,100). If the starting index = 1 and the end index = 3 then the polygon will be delimited by the following points:

\[
(110,100) - (100,115) - (90,100) - (110,100)
\]

The last point has been added automatically to complete the polygon.
A special point, **nullPoint** ((-32768,-32768), see "MG - Points" under "Constants"), is treated differently so that **FillPolygon** stops drawing the polygon in progress and starts a new polygon using the point after **nullPoint**. **nullPoint** is used to fill several separate polygons and to prevent unwanted parts of a polygon being filled. If two polygons are completely disjoint, both will then be filled. If one polygon lies inside the other then the area **between** the polygons will be filled. This can be done with a single call to **FillPolygon**, which is very useful when drawing symbols and signs.

Using the same array of points above, but with a new end index = 7, the area between the following polygons is then delimited and filled in:

\[(110,100) - (100,115) - (90,100) - (110,100)\] and
\[(100,105) - (100,110) - (95,105) - (100,105)\]

*Integer overflow in additions of the point data with the current position is neither checked, nor reported!*

**SV.Result**
- **processWithoutWin** the calling process has no window
- **deviceProcNotInUse** the driver does not support the procedure
- **invalidPolygon** at least 3 points/end index >= the starting index

Errors that are specific for each driver are also reported.

**TYPE**
```
PolyArray = ARRAY[0..11] OF MG.Point;
CONST
minInt = -32768;
pa : PolyArray = (
  (x: 0;y: 0),
  (x:30;y: 0),
  (x:30;y:30),
  (x: 0;y:30),
  (x: 0;y: 0),
  (x:minInt;y:minInt),
  (x:10;y:10),
  (x:20;y:10),
  (x:20;y:20),
  (x:10;y:20),
  (x:10;y:10),
  (x:minInt;y: minInt));
```
MGPat.SetGenPattern(MGPat.vert);
MG.SetPos(400,200);
MG.FillPolygon(pa,0,11);
(*Fills the area between the large and the small square*)

MG.SetPos(500,200);
MG.FillPolygon(pa,0,5); (*Fills the large square*)
MG.SetPos(600,200);
MG.FillPolygon(pa,5,11); (*Fills the small square*)

**FillShortPolygon**

P FillShortPolygon(VAR vertices; (*I*)

    first, last: Word);

This procedure fills a polygon using the current position and the array of relative point data which is found in the variable parameter vertices. The polygon starts from the point with the index first and ends at the point with the index last, where last \(\geq\) first. If the endpoint does not coincide with the starting point then the polylines are automatically joined to form a polygon.

The polygon is clipped against the current viewport and is filled using the current window mode, i.e. the foreground and background color, color mask and drawing mode. No border line is drawn.

The current position and drawing direction are not changed.

vertices must be of the type ARRAY [0..maxindex] OF ShortPoint, that is to say an array with a starting index 0 and a maximum index, maxindex, which depends on the number points in the array. The number of points must be at least 3 and can be at most 1/8 of bufferSize (750 points if the buffer contains 6000 bytes). The elements of the array must be of the type ShortPoint (see "MG - Points" under "Data types").

The coordinates of the points are not given as absolute coordinates, but are instead relative to the current window position.

Suppose that we have declared an array of points as follows:

\[
((0,5),(10,0),(0,15),(-10,0),
(-128,-128),(0,5),(0,10),(-5,5))
\]

and the current position is (100,100). If the starting index = 1 and the end index = 3 then the polygon will be delimited by the following points:
The last point has been added automatically to complete the polygon.

A special point, nullShortPoint ((-128,-128), see "MG - Points" under "Constants"), is treated differently so that FillShortPolygon stops drawing the polygon in progress and starts a new polygon using the point after nullShortPoint. nullShortPoint is used to fill several separate polygons and to prevent unwanted parts of a polygon being filled. If two polygons are disjoint, both will then be filled. If one polygon lies inside the other then the area between the polygons will be filled. This can be done with a single call to FillShortPolygon, which is very useful when drawing symbols and signs.

Using the same array of points above, but with a new end index = 7, the area between the following polygons is then delimited and filled in:

\[(110,100) - (100,115) - (90,100) - (110,100)\]
\[(100,105) - (100,110) - (95,105) - (100,105)\]

Integer overflow in additions of the point data with the current position is neither checked, nor reported!

<table>
<thead>
<tr>
<th>SV.Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>processWithoutWin</td>
<td>the calling process has no window</td>
</tr>
<tr>
<td>deviceProcNotInUse</td>
<td>the driver does not support the procedure</td>
</tr>
<tr>
<td>invalidPolygon</td>
<td>at least 3 points/end index &gt;= the starting index</td>
</tr>
<tr>
<td>tooManyPoints</td>
<td>max. number 1/8 of bufferSize</td>
</tr>
</tbody>
</table>

Errors that are specific for each driver are also reported.

Filling Circles and Sectors of Circles

```
P FillCircle(radius: INTEGER);
```

This procedure fills a circle using the radius radius, with the center at the current position and using the current pattern. The radius is given in x units.

The circle is clipped against the current viewport and is filled using the cur-
rent window mode, i.e. using the foreground and background colors, color mask and drawing mode. No circumference line is drawn.

The current position and direction are not modified.

**SV.Result**
- **processWithoutWin** the calling process has no window
- **invalidRadiusOrAxis** the radius must be > 0

Errors that are specific for each driver are also reported.

```pascal
MG.SetPos(200,200);
MGPat.SetGenPattern(MGPat.solidPat);
MG.FillCircle(100);
```

**FillSector**

```pascal
P FillSector(radius, centralAngle: INTEGER);
```

This procedure fills a sector of a circle with the current pattern. The center is at the current window position, the starting angle is the same as the current drawing direction, the radius is `radius` and the central angle is `centralAngle`, measured in tenths of degrees (decidegrees). The radius is specified in x units.

![Fig 1: Circle sector](image)

A positive central angle draws the sector *counter-clockwise*, while a negative angle draws it clockwise.
The sector is clipped against the current viewport and is filled using the current window mode, i.e. foreground and background colors, color mask and drawing mode.

The current direction is assigned to the final direction of the sector's arc. The direction is normalized within the range of 0..3599 decidegrees. The current position is not modified on the other hand.

\[ \text{SV.Result} \]
\[ \text{processWithoutWin} \quad \text{the calling process has no window} \]
\[ \text{invalidRadiusOrAxis} \quad \text{the radius must be } > 0 \]

Errors that are specific for each driver are also reported.

MG.SetPos(200,200);
MG.MatchForeground(MG.lightGreen);
MG.MatchBackground(MG.green);
MGPat.SetGenPattern(MGPat.diagCross);
MG.SetDir(1350);
MG.FillSector(100,-450);

Filling Ellipses and Sectors of Ellipses

\[ \text{FillEllipse} \]
\[ \text{FillEllipse}(a,b,\text{majorAxisAngle}: \text{INTEGER}); \]

This procedure fills an ellipse with the current pattern. \( a \) specifies the ellipse's major axis and \( b \) its minor axis. \text{majorAxisAngle} specifies the direction of the major axis in tenths of degrees (decidegrees). The center of the ellipse is at the current position. The major and minor axes are specified in \( x \) units.
The ellipse is clipped against the current viewport and is filled using the current window mode, i.e. foreground and background colors, color mask and drawing mode.

The current position and direction are not modified.

**SV.Result**

- `processWithoutWin` the calling process has no window
- `invalidRadiusOrAxis` minor and major axis must be > 0

Errors that are specific for each driver are also reported.

```pascal
MG.SetPos(200,200);
MGPat.SetGenPattern(MGPat.halftone8);
MG.FillEllipse(100,50,450);
```

### FillEllipseSector

```pascal
P FillEllipseSector(a,b,majorAxisAngle: INTEGER;
   centralAngle : INTEGER);
```

This procedure fills an *ellipse sector* with the current pattern. The center of the ellipse is at the current window position, its starting angle is the same as the current drawing direction, the major axis is given in `a`, the minor axis in `b`, the direction of the major axis in `majorAxisAngle` and the central angle in `centralAngle`, measured in tenths of degrees (decidegrees). The major and minor axes are specified in x units.

A positive central angle draws the sector counter-clockwise, while a nega-
tive draws it clockwise.

The ellipse sector is clipped against the current viewport and is filled using the current window mode, i.e. foreground and background colors, color mask and drawing mode.

The current direction is assigned to the final direction of the sector's arc. Direction is normalized within the range of 0..359.99 degrees. The current position, on the other hand, is not modified.

SV.Result
processWithoutWin the calling process has no window
invalidRadiusOrAxis the axis must be > 0

Errors that are specific for each driver are also reported.

MG.SetPos(200,200);
MG.MatchForeground(MG.green); MG.MatchBackground(MG.white);
MG.SetDir(450);
MGPat.SetGenPattern(MGPat.horiz);
MG.FillEllipseSector(100,50,450);

Filling of Polybézier Curves

FillPolyBezier

Fill a polybézier curve relative the current position and the array of relative Bézier elements found in the variable parameter polyBez. The polybézier starts from the Bézier element with the index first and ends with the element with the index last, where last >= first. The filled polybézier is clipped against the current viewport and is drawn using the current window mode, i.e. foreground and background colors, color mask, drawing mode, line style, line style position, line style protection, line width, and endpoint skip mode.

The border line is also drawn (in contrast to FillPolygon), if the line style is not set to nullLineStyle.

The current position and drawing direction are not changed.
The only permitted start index of the polybézier array is 0. The coordinates of the control points do not specify absolute coordinates, instead these are given relative to the current window position.

SV.Result
processWithoutWin the calling process has no window
invalidPolygon last has to be >= first
tooManyPoints internal buffer filled up

Errors are also reported that are specific for each respective driver.

TYPE
PolyBez = ARRAY[0..4] OF MG.Bezier;
CONST
pb: PolyBez = (((x:10;y:250),(x:50;y:400),
(x:250;y:400),(x:300;y:250)),
((x:300;y:250),(x:250;y:50),
(x:50;y:50),(x:10;y:250));

MG.SetPos(50,0);
MG.FillPolyBezier(pb,0,1);

Filling an Irregular Area

Flood fills an irregular area in the current window with the current pattern. The flooding starts from the current position and continues until one of the following alternative criterions holds:

1) If the color of the current position is separate from borderCol the filling continues until points with the device color in borderColor are encountered, or until the flooding reaches the current viewport.

2) If the color of the current position is the same as borderCol the filling continues until points with a different device color are encountered, or until the flooding reaches the current viewport.
FillArea acts solely on screens where existing graphics on the screen can be read or where there is hardware support for flood filling areas.

SV.Result
processWithoutWin the calling process has no window
outsideView the current position must be inside the current viewport
deviceProcNotInUse the driver has no support for the procedure

Errors that are specific for each driver are also reported.

MG.SetPos(100,100); MGPat.SetGenPattern(MGPat.cross);
MG.MatchForeground(MG.yellow); MG.MatchBackground(MG.blue);
MG.FillArea(MG.MatchingForeground(MG.white));
MultiGraphics contains procedures (CopyView and CopyRect) that copy graphics from one rectangular area to another. Graphics can be copied

- within the same window,
- between windows in the same screen,
- between screens in the same device and
- between different devices.

However, drivers are allowed to restrict the copying capabilities.

Using the copying routines, it is possible to copy graphics from the visible screen to virtual screens and vice versa. Copying can be used for quick animated graphics etc.

With printing devices it is common that the visible screen corresponds to the paper. Copying from a virtual screen to the visible screen has then the effect of printing the hidden graphics in the virtual screen on the paper.

MultiGraphics also contains "low level" procedures (ReadBitBlock and DrawBitBlock) that copy rectangular screen areas to and from monochrome bit blocks in memory.

Overview of Copying

Copying (using CopyView or CopyRect) is always carried out from one specified window, the source window, to the current window. The source area is copied to the current position. If the source area is in the current
screen the destination area is allowed to overlay the source area.

Copying is carried out using only those device colors in the original that are both let through the color mask in the original window and through the color mask in the current window. The drawing mode in the current window is used.

If the source screen is monochrome, the foreground and background colors in the current window are used. Those parts of the source that are set to zero are copied using the current background color, while those parts set to one are copied using the foreground color.

If the source screen contains color information and the current screen is monochrome, those parts of the original that have the current background color in the source are copied as if they had the device color 0, while parts with other colors are copied as if they had device color 1.

Copying between Different Devices

When copying between different devices certain requirements regarding compatibility are placed on the devices. The following conditions must be met:

- the original screen must be of the virtual or monochrome virtual type (i.e. not a visible screen nor a special screen),
- both the devices must be of the raster type,
- both the screens must have the same pixel representation (the number of bits per pixel),
- if the original screen is a virtual screen and the current screen is visible, special or virtual (not monochrome), the original screen is not allowed, after color masking, to contain device colors that are not to be found in the color mask of the current screen.

For example, the last requirement does not allow copying between an original screen whose color mask lets through device color 8 (color plane 3 if plane oriented) to a current screen whose color mask is 7 (color planes 0, 1 and 2).

The requirement of having the same pixel representation, for example, would rule out copying between MCGA (byte oriented) and VGA (bit oriented) in color mode.

When copying from a visible screen or special screen in one device to a screen in another device, the copying is first made to a temporary virtual screen, created and disposed of by the copy procedure. This virtual screen, which never holds more than a few
lines, is created in the current virtual memory (conventional, extended XMS, expanded EMS or disk).

**Hard Copy**

The copying procedures in MG can be used to make hard copies of screens on printers. Here is an outline of a program that makes a hard copy of the visible screen:

```pascal
VAR
   scrDrv, prDrv: MG.Device;
   scrWin      : MG.Window;
..
MG.SetDevice(scrDrv);
scrWin:= MG.CurWin;
MG.SetDevice(prDrv);
(* Print black as white, and white as black *)
MG.SetRGB(0, MG.white);
MG.SetRGB(MG.CurMaxDeviceColor, MG.black);
MG.SetPos(..); (* Printing position *)
MG.CopyView(scrWin); (* Printout *)
..
MG.FlushVisiScr; (* Feed paper *)
```

**Bit Blocks**

It is sometimes necessary to read and write bit blocks, i.e. patterns of "zero" bits and "one" bits. Rectangular bit blocks are stored row-by-row as bits in conventional memory. Each row holds an integer number of bytes (8 bits). However, when using `DrawBitBlock` for drawing it is not necessary to draw the entire bit block, instead you can draw a smaller part of it.

Bits in the block that are set to 1 are drawn using the foreground color of the current window and bits that are set to 0 are drawn using the current background color. The lower, left corner of the block is positioned at the current window position.

When you read bit blocks using `ReadBitBlock`, many bit oriented device drivers as VGA, EGA, Hercules, etc., avoid to shift the bit pattern for the sake of performance. Therefore the bit pattern that has been read is often shifted in the horizontal direction (up to 7 bits).

The block is read from the current position. The current color mask of the
current window is used to translate the read pixels to "0" and "1". Colors corresponding to the color mask are translated to "1" and all other colors to "0".

The bit block operations are also affected by the current drawing mode of the current window.

Bit block operations are the secret behind both bitmapped characters and symbols in *MultiGraphics* (see "MG - Symbols" and "MG - Fonts") and bit blocks are also used to compress and decompress PCX images. Generally speaking, bit blocks are good for transferring external bitmapped information to and from *MultiGraphics* screens.

### Examples

**GACopy** makes use of block copy operations to move an object across the screen. **GAHCopy1** makes a screen "dump" to an Epson, IBM or NEC printer. **GAHCopy2** makes a hardcopy to HP LaserJet. **GAHCopy4** makes a hardcopy to HP DeskJet 500C or 550C. **GABBlock** defines and draws a simple bit block.

### Procedures & Functions

#### Copying Routines

<table>
<thead>
<tr>
<th>CopyRect</th>
<th>DLR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P CopyRect</strong> (source : Window;</td>
<td></td>
</tr>
<tr>
<td>x0,y0 : INTEGER;</td>
<td></td>
</tr>
<tr>
<td>width,height : INTEGER);</td>
<td></td>
</tr>
</tbody>
</table>

This procedure copies the graphics in a rectangular area, located in the window `source`, to the current window. The original rectangle's lower left corner is specified using `(x0,y0)` in `source` window coordinates. The width and the height are given by `width` and `height` respectively. The copy's lower left corner is placed at the current position and window.
The rectangle is clipped first against the original window's viewport and then against the current window's viewport (the clipping does not affect the position).

Otherwise, what has been said previously in this chapter applies.

SV.Result
processWithoutWin calling process has no window
invalidWin invalid source window
incompatibleDevices non-compatible windows in different devices
deviceProcNotInUse the driver does not support this procedure

Errors that are specific for the current device are also reported.

VAR
x,y: INTEGER;
...
MG.SetPos(35,25); MGPat.SetGenPattern(MGPat.horiz);
MG.MatchForeground(MG.white);
MG.MatchBackground(MG.blue);
MG.FillEllipse(30,20,0);

x:= 0; y:= 0;
REPEAT
MG.SetPos(x + 4,y + 4);
MG.CopyRect(MG.CurWin,x,y,70,50);
INC(x,4); INC(y,4);
UNTIL (x >= MG.CurWinWidth) OR
(y >= MG.CurWinHeight);

The above example shows that a rectangle can copy itself onto itself - here we are using it for animation.
**CopyView**

```pascal
P CopyView(source: Window);
```

This procedure copies the graphics in the viewport, located in the `source` window, to the current window. The copy's lower left corner is placed at the current position in the current window.

When the copying is carried out, the graphics is clipped against the current window's viewport.

[Diagram showing the process of copying graphics]

Fig 1: Copying graphics

Otherwise, what has been said previously in this chapter applies.

**SV.Result**

- **processWithoutWin** calling process has no window
- **invalidWin** invalid `source` window
- **incompatibleDevices** non-compatible windows in different devices
- **deviceProcNotInUse** the driver doesn't support this procedure

Errors that are specific for the current device are also reported.
VAR
  monoDev, colorDev: MG.Device;
  virtScr : MG.Screen;
  monoWin, virtWin : MG.Window;

  MG.SetDevice(monoDev);
  MG.SetVisiScr; MG.SetView(50,50,200,100);
  monoWin := MG.CurWin;
  MG.CreateMonoVirtScr(virtScr, 200, 100);
  MG.SetScr(virtScr);
  MG.CopyView(monoWin);
  virtWin := MG.CurWin;
  MG.SetDevice(colorDev);
  MG.SetPos(50,50);
  MG.SetForeground(10); MG.SetBackground(1);
  MG.CopyView(virtWin);
  MG.CloseScr(virtScr);

The above example shows how you can copy the visible screen of a monochrome device to the visible color screen of another device. Because we cannot copy directly between visible screens in different devices, the copying is done via a mono virtual screen.

<table>
<thead>
<tr>
<th>ZoomVirtScr</th>
<th>DLR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>See &quot;MG - Screens&quot;.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DrawBitBlock</th>
<th>DLR</th>
</tr>
</thead>
</table>
| P DrawBitBlock(VAR sourceBitBlock (*I*)
  rowSize : INTEGER;
  leftSkip, topSkip: INTEGER;
  width, height : INTEGER); |
|              |     |
| This procedure draws the bit block in sourceBitBlock, at the current position in the current window, using the current drawing mode, foreground and background colors. |
rowSize specifies the number of bytes in each row of sourceBitBlock. leftSkip contains the number of bits (>= 0) to skip from the left of the bit block. topSkip tells the number of rows (>= 0) to skip from the top of the block. width (>= 0) specifies the number of bits to draw and height (>= 0) the number of rows.

Bits set to "1" are painted using the current foreground color of the destination window, while bits set to "0" bits are painted using the current background color.

SV.Result
calling process has no window
deviceProcNotInUse the driver doesn't support this procedure
invalidBlockSize invalid leftSkip, topSkip, width or height parameter

TYPE
  BitBlock = ARRAY[0..3] OF ARRAY[0..1] OF Byte;
CONST
  block: BitBlock = (($F0,$01),
                   ($80,$01),
                   ($80,$01),
                   ($80,$0F));
...
MG.DrawBitBlock(block,2,0,0,16,4);
ReadBitBlock

P ReadBitBlock(VAR destBitBlock; (*O*)
    width, height : INTEGER;
    VAR rowSize : INTEGER; (*O*)
    VAR leftSkip, topSkip : INTEGER (*O*);)

This procedure reads pixel information from the specified rectangle area of the current window and the data is stored in the bit block destBitBlock. The lower left corner of the rectangle is located at the current window position, while width and height specify the pixel size of the rectangle. Pixel colors matching the current color mask (see MG.SetColorMask) are stored as "1" bits while all other colors are stored as "0" bits.

The number of bytes per row is returned in the rowSize parameter. The bit data stored may be shifted from the left with leftSkip bits and from the top with topSkip lines.

In fact topSkip always returns 0 because all supported devices today are row oriented. leftSkip, however, may be > 0, when reading from bit oriented devices and screens, for example CGA, EGA, VGA, Hercules and monochrome virtual screens. leftSkip is 0 when reading from byte oriented devices. Therefore it is a good advice to consider the information in leftSkip.

SV.Result
calling process has no window
the driver doesn't support this procedure
invalid width or height parameter
specified block is not fully visible
invalid color mask

VAR
driver: MG.Device;
bitblock: ARRAY[0..99] OF Byte;
rowSize,
leftSkip,
topSkip : INTEGER;
.. MG.SetPos(3,3);
MG.DrawBox(6,6);
MG.SetPos(3,3);
MG.SetColorMask(MG.CurMaxDeviceColor)
MG.ReadBitBlock(bitblock,6,6,rowSize,leftSkip,topSkip);
MG.SetPos(0,0);
MG.DrawBitBlock(bitblock,rowSize,leftSkip,topSkip,6,6);
Executing this code on a bit oriented device or a mono virtual screen, **ReadBitBlock** returns the following values:

- rowSize = 2 (bytes)
- leftSkip = 3
- topSkip = 0

Running on a byte oriented device or screen, **ReadBitBlock** returns the following values:

- rowSize = 1 (bytes)
- leftSkip = 0
- topSkip = 0

The same data can be used when calling **DrawBitBlock**.
MultiGraphics supports *symbol libraries* and *symbol drawing*. Each device driver usually defines a set of *standard symbols*, convenient for use as markers and mouse pointers. Other symbols can be created anew or copied and modified from other symbol libraries and eventually stored in symbol files. Symbol libraries are attached to variables of the type *Symbols* and can store up to 256 different symbols, each one of varying size and appearance. Symbol libraries can be shared between windows, but each window can have its own *current* symbol library.

Symbols can be used as cursors, mouse pointers, icons, CAD symbols etc.

Symbols are always *monochrome* and therefore are drawn using the current foreground and background colors. There are four different *symbol types*, namely:

- bitmapped symbols (*bitmapSymbols*),
- stroke symbols (*strokeSymbols*),
- outline symbols (*outlineSymbols*) and
- Bézier symbols (*bezierSymbols*)

Stroke, outline and Bézier symbols can be transformed, i.e. be scaled, rotated and sheared.

Shearing means that a symbol is deformed in the x or y direction. Italicized characters are examples of symbols that have been "sheared".

Symbols can be transformed once and for all in a way that is independent of output devices, screens and windows, called *global* transformation, and/or they can be transformed *locally* in each window in conjunction with symbol
Symbols are related to fonts. The exact relationship will be described later.

Global Resources

Symbol libraries are *global resources*. This means that they are created, loaded and deleted independently of the active devices and may be shared by several devices, screens and windows. This also means that symbols must be deallocated manually from memory by calling `MG.DisposeSymbols`. They are not disposed of automatically when closing device drivers.

An exception from this is the built-in default symbol library supplied with the device driver. This is removed from memory together with the driver.

Current Symbols

Each window can have its own current symbol library. When a window is created the default symbols of the device are installed.

Symbol Types

`MultiGraphics` accepts four different symbol types:

![Symbols](image)

Fig 1: Bitmapped, stroke, outline and Bézier symbols
Bitmapped Symbols

Bitmapped symbols are made up of bit block patterns. Their main advantages are:

+ high speed,
+ exact control of symbol appearance.

Bitmapped symbols can often be drawn at high speeds, especially if the output is restricted to "speed positions" (see GetFastXY in MG). Stroke, outline and Bézier symbols can be converted to bitmapped symbols.

The disadvantages of bitmapped symbols are:

- cannot be scaled, rotated or sheared,
- can only be used with raster devices,
- take up a lot of memory when large symbols are being used,
- drawing speed is reduced by a power of two to the symbol dimensions.

Stroke Symbols

Stroke symbols are made up of straight line segments, which form one or more polylines. The advantages of this type of symbol are:

+ can be scaled, rotated and sheared,
+ work on all types of devices,
+ memory requirement is not dependent on symbol size,
+ drawing speed is reduced linearly compared to symbol dimensions.

The disadvantages are:

- they are usually slower than small bitmapped symbols,
- they may look "sparse" because the symbols are constructed with lines,
- they need lots of points to create a smooth appearance,
- they appear jagged when reduced,
- they appear unsmoothed when enlarged too much.

Outline Symbols

Outline symbols are constructed with line segments, in the same way as stroke symbols. In contrast to stroke symbols, however, the line segments always form closed polygons, which are filled using the current pattern. If
the current pattern is **nullPattern**, filling is not carried out. The outline lines are drawn at the same time as the symbols. If the current line style is **nullLineStyle** the outline is not drawn. The advantages of outline symbols are:

+ can be scaled, rotated and sheared,  
+ can be filled with patterns which means that symbols fill up the drawing area better,  
+ can be used on all types of devices that support polygon filling.

The disadvantages include:

- they are slower than stroke symbols and bitmapped symbols,  
- they require support for polygon filling,  
- they need lots of points to create a smooth appearance,  
- they appear jagged when reduced,  
- they appear unsmoothed when enlarged too much.

**Bézier Symbols**

Bézier symbols are constructed with Bézier curve elements. Bézier curves usually belong to a class of third degree polynomials with properties that make them appropriate for description of symbols and characters. In contrast to the straight edges of outline symbols, points in Bézier outlines can be computed to any level of precision, which makes the appearance smooth.

Bézier symbols always form closed curves, which are filled using the current pattern. If the current pattern is **nullPattern**, filling is not carried out. The outline curve is drawn at the same time as the symbols. If the current line style is **nullLineStyle** the outline is not drawn. The advantages of Bézier symbols are:

+ can be scaled, rotated and sheared and still appear smooth,  
+ can be filled with patterns,  
+ can be used on all types of devices that support polygon filling.

The disadvantages include:

- they are slower than outline, stroke and bitmapped symbols,  
- they require support for polygon filling,  
- they appear jagged when reduced too much.
Conversion to Bitmapped Symbols

You may convert scaled stroke, outline and Bézier symbols to bitmapped symbols by calling `CreateBitmapSymbols`. The converted symbols can then be saved in a symbol file by calling `SaveSymbols`. This approach combines the best of two worlds, namely the speed of bitmapped symbols and the flexibility of vector and Bézier symbols when scaling and doing patterns.

Standard Symbols

Most drivers define a number of *standard symbols* to be used with the device in question. The following symbol numbers are reserved:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Face</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursors:</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>plus</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>star</td>
</tr>
<tr>
<td>3</td>
<td>square</td>
</tr>
<tr>
<td>4</td>
<td>filled square</td>
</tr>
<tr>
<td>5</td>
<td>rhombus</td>
</tr>
<tr>
<td>6</td>
<td>circle</td>
</tr>
<tr>
<td>..</td>
<td></td>
</tr>
<tr>
<td>Mouse pointers:</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>arrow left upwards</td>
</tr>
<tr>
<td>65</td>
<td>arrow up</td>
</tr>
<tr>
<td>66</td>
<td>arrow down</td>
</tr>
<tr>
<td>67</td>
<td>arrow left</td>
</tr>
<tr>
<td>68</td>
<td>arrow right</td>
</tr>
<tr>
<td>69</td>
<td>cross</td>
</tr>
<tr>
<td>70</td>
<td>x</td>
</tr>
<tr>
<td>71</td>
<td>vertical bar</td>
</tr>
<tr>
<td>72</td>
<td>check mark</td>
</tr>
<tr>
<td>73</td>
<td>hand</td>
</tr>
<tr>
<td>74</td>
<td>hour glass</td>
</tr>
<tr>
<td>75</td>
<td>vertical dual arrows</td>
</tr>
<tr>
<td>76</td>
<td>horizontal dual arrows</td>
</tr>
<tr>
<td>77</td>
<td>dual arrows diagonal down</td>
</tr>
<tr>
<td>78</td>
<td>dual arrows diagonal up</td>
</tr>
<tr>
<td>79</td>
<td>arrow cross</td>
</tr>
</tbody>
</table>
Cursor masks:

128  plus
129  x
130  star
131  square
132  filled square
133  rhombus
134  circle

Mouse pointer masks:

192  arrow left upwards
193  arrow up
194  arrow down
195  arrow left
196  arrow right
197  cross
198  x
199  vertical bar
200  check mark
201  hand
202  hour glass
203  vertical dual arrows
204  horizontal dual arrows
205  dual arrows diagonal down
206  dual arrows diagonal up
207  arrow cross

Symbol numbers in the interval 128..255 (i.e. bit 7 is on) are used as masks for the symbols 0..127.

The symbol 66 has a corresponding mask with number 66 + 128 = 194.

The driver's standard symbols are set as the current symbols when a window is created. They are also set by calling `SetDefaultSymbols`. See the documentation for each driver for more details.

Symbol Information

Each individual symbol is characterized by the value of its width, height and "hot spot" (drawing point). All three features are unique for the symbol.
The values are affected by the current transformation.

![Symbol information diagram](image)

**Fig 2: Symbol information**

The *hot spot* marks the point in the symbol that is placed at the current position when the symbol is drawn.

**Large and Small Symbols**

Bitmapped, stroke and outline symbols can use two different symbol formats, namely:

- small symbols with a maximum width and height <= 127 pixels,
- large symbols with a maximum width and height <= 32767 pixels.

These limits must not be exceeded when you transform symbols (stroke and outline symbols).

Small symbols require less memory, while large symbols allow more options when scaling. When drawing on devices with high resolutions, such as laser printers and pen plotters, it may be necessary to use large symbols.

Bézier symbols are always of the "large" symbol size.

Each symbol format has its own file format.
Symbol Files

MultiGraphics has its own unique symbol file format. A symbol file can store up to 256 symbols. All the symbols of a library are of the same type (bitmapped, stroke, outline or Bézier). Besides the description of the individual symbols, the symbol file also contains information about the width and height for individual symbols, as well as the "hot" point for each symbol etc.

We recommend using the following file extensions for symbol files:

- **SBL** "Symbols Bitmapped Large"
- **SBS** "Symbols Bitmapped Small"
- **SSL** "Symbols Stroke Large"
- **SSS** "Symbols Stroke Small"
- **SOL** "Symbols Outline Large"
- **SOS** "Symbols Outline Small"
- **SZL** "Symbols Bézier Large"

Plain font files use "C" instead of "S" as a first letter in the file extension. See "MG - Fonts" for details.

Creating and Modifying Symbols

In MG you will find procedures for creating new symbol libraries, modifying old libraries and saving the libraries on disk. Symbol data can be read from existing libraries.

A new and empty symbol library is created by calling **CreateSymbols**. The new library is of a specific type (bitmapped, stroke, outline or Bézier) and size ("small" or "large") and is moreover characterized of having fixed or non-fixed symbols widths, heights and hot spots.

Bézier symbols can only have "large" symbols.

You can add new symbols, or change old symbols in a symbol library by calling **AddSymbol**. The symbol data is dependent on the type of symbol as follows:

Bitmapped symbols are described by a bit block (see "MG - Copy Graphics"), where the bits set to "1" are used for drawing pixels and bits set to "0" for not drawing.
Stroke and outline symbols are described by an array of points. "Small" symbols use points of the type MG.ShortPoint, while "large" symbols use points of the MG.Point type. MG.nullPoint and MG.nullShortPoint are used to create "jumps" in the curve (see DrawPolyline and DrawShortPolyline in "MG - Lines" and FillPolygon and FillShortPolygon in "MG - Patterns").

Bézier symbols are described by an array of Bézier elements of the type MG.Bézier, elements that in turn are defined by an array of four control points of the MG.Point type. The curve is continuous if the endpoint of one Bézier element is the starting point of the next element. Otherwise there will be a "jump" in the curve (see DrawPolyBez in "MG - Lines" and FillPolyBezier in "MG - Patterns").

You can get all information about a symbol, in an existing symbol library, by calling GetSymbolData or GetSymbolDataPtr. This will return information on the symbol width, height, hot spot and data (the structure is dependent on the type of symbol library). The structure of the information received is the same as the structure of the information fed to AddSymbol, so that you can use the information returned as input to AddSymbol.

The difference between GetSymbolData and GetSymbolDataPtr is that the first procedure puts data in an existing buffer supplied by the caller, while the second procedure allocates a new data buffer on the program heap and returns a pointer to the data.

Symbols can be deleted from a symbol library by calling DeleteSymbol. Symbol libraries can be stored permanently in symbol files on disk, by calling SaveSymbols. They can be loaded by calling LoadSymbols.

Taken as a whole the procedures in this overview can be used to create completely new symbol libraries from scratch, they can be used to modify and cut from old libraries or to borrow symbols from one library to another. The result can be stored in symbol files on disk.
Transformations

Symbols of the stroke, outline or Bézier type can be scaled, rotated and sheared.

*Scaling* means that something is changed in size. When scaling is carried out, the x and y values are multiplied by scale factors. The x and y coordinates can have different scale factors and these factors are specified in units of parts per thousand, which means that the value of 1000 corresponds to a scale of 1:1, i.e. no change in size. The scale factor of 2000 means doubling the scale, and so on.

*Rotation* is specified by an angle given in tenths of degrees (decidegrees). 10 decidegrees correspond to 1 degree. 3600 decidegrees correspond to 360 degrees, i.e. one complete revolution. Positive angles give an anti-clockwise rotation while negative angles rotate clockwise.

*Shearing* means that symbols are deformed ("slanted") in the x or y direction. The x and y directions can have different shearing factors. Shearing is specified in units of parts per thousand, which means that a value of 1000 corresponds to a factor of 1. The value 0 entails that no shearing occurs at all. When shearing takes place in the x direction, the symbols are "italicized".

Global Transformation of Symbols

Stroke, outline and Bézier symbols can be transformed *globally* once and for all, and independently of the output device, screen and window by transforming the complete description of the symbol attached to a variable of the type *Symbols*. No time consuming calculations need therefore be carried out before characters are printed out.

Globally transformed symbol libraries can be transformed yet again, but repeated transformations can lead to rounding off errors. If you need to use the same symbol library with different transformations it is better to load the symbol file again, or load the symbol file several times and attach it to different *Symbols* variables.

Local Window Oriented Transformation

Stroke, outline and Bézier symbols can also be transformed *locally* in each window. Each window can have its own current transformation. The trans-
formation is applied first when the printout is done, which means that performance cannot be as high as with global transformations. On the other hand, flexibility is increased and memory requirements are decreased because the symbol library only needs to be loaded once.

If a symbol library has been globally transformed, then the window oriented transformations affect the library's transformed symbols, not the original symbols.

When a window is created, the window transformation is switched off.

Symbols and Fonts

Symbols and fonts are closely related. In reality, each font file, except device specific font files (of the type deviceSymbols), can also be loaded and used in the same way as symbol files. The opposite, however, does not always apply - in general, a symbol file cannot be loaded in the same way as a font file.

Symbols and fonts use the same representations. These can be of four types, bitmapped (bitmapSymbols), stroke (strokeSymbols), outline (outlineSymbols) or Bézier (bezierSymbols).

The only difference is the following: characters in a font can have different widths but must have the same height and base line. This restriction does not apply to symbols - individual symbols can have different heights and hot spots (a broader concept than the term base line).

A font file can therefore always be loaded in the same way as a symbol file (if it is not a device specific font).

Examples

GASym shows scaling, rotation and shearing of symbols. GAMkSym gives an example of the creation of a simple symbol library.
SymbolsType = (bitmapSymbols,
              strokeSymbols,
              outlineSymbols,
              bezierSymbols,
              deviceSymbols);

Symbol and font types in MultiGraphics. These types are described above.

SymbolSet = SET OF 0..maxSymbol;

Symbols set type.

SymbolsInfo = RECORD
  symbType : SymbolsType;
  symbolsId : Word;
  deviceId : Word;
  deviceFamily : Word;
  first, last : Word8;
  fixedWidth : INTEGER;
  maxWidth : INTEGER;
  averageWidth : INTEGER;
  maxX : INTEGER;
  fixedHeight : INTEGER;
  maxHeight : INTEGER;
  averageHeight : INTEGER;
  maxY : INTEGER;
  fixedHotSpot : Point;
  lowLeftPoint : Point;
  shortSize : BOOLEAN;
  ..
END;

Type used with GetSymbolsInfo and GetCurSymbolsInfo.
## Constants

\[
\text{minSymbolsType} = \text{bitmapSymbols}; \\
\text{maxSymbolsType} = \text{deviceSymbols}; \\
\]

These are the constants for the first (\text{bitmapSymbols}) and last (\text{deviceSymbols}) values in the enumerated data type \text{SymbolsType}.

\text{nullSymbols}

Specifies a non-existing symbol library.

\[
\text{maxSymbol} = 255; \\
\text{allSymbols} = [0..\text{maxSymbol}]; \\
\]

Defines the maximum symbol value and a set constant to be used when creating bitmapped symbols with \text{CreateBitmapSymbols}.

## Variables

\text{maxRamSymbolsSize}: \text{Word};

This variable specifies the maximum byte size for a symbol library or a font that can be loaded into program heap memory using \text{LoadSymbols} and \text{LoadFont}.

If the symbol library or the font is larger than the value in \text{maxRamSymbolsSize}, the symbols and fonts will instead be loaded into the current virtual memory (set by \text{VM.SetVM} or automatically by \text{AutoVM}).

When the unit \text{MG} is initialized, \text{maxRamSymbolsSize} is set to 5000 bytes in real mode and 65535 bytes in protected/extended mode.
Creating, Loading and Saving Symbols

**CreateSymbols**

```pascal
P CreateSymbols(VAR symbs : Symbols; (*O*)
    symbsType : SymbolsType;
    shortSym : BOOLEAN;
    fixWidth : INTEGER;
    fixHeight : INTEGER;
    fixHotSpot : Point);
```

This procedure creates a new and empty symbol library and attaches it to the variable `symbs`. The type of symbol library is given in `symbsType` (bitmapped, stroke, outline or Bézier type). If `shortSym` is set to `TRUE` a library of "small" symbols (max. 127x127 pixels) is created and `FALSE` creates a library of "large" symbols.

- `fixWidth >= 0` creates a library of fixed width symbols. `fixWidth <= 0` creates a library where the width of the symbols can vary.
- `fixHeight >= 0` creates a library of fixed height symbols. `fixHeight <= 0` creates a library where the height of the symbols can vary.
- `fixHotSpot <> MG.nullPoint` creates a library of fixed hot spot positions. `fixHotSpot = MG.nullPoint` creates a library where the hot spot position of the symbols can vary.

The new symbol library is created in the current virtual memory and has an offset table of maximum length that can store up to 255 symbols. New symbols are added by calling `AddSymbol`.

A symbol library can be saved in symbol files on disk by calling `SaveSymbols`. When the symbol library is saved, the offset table is minimized. When the library is reloaded from disk using `LoadSymbols`, the library will therefore usually need less heap space than before.
SV.Result
MGErr.outOfMem not enough memory to create the library
MGErr.invalidSymbols MG.deviceSymbols not allowed

Errors can also be reported from the virtual memory handler in VM.

VAR
    symbs: MG.Symbols;
    poly : ARRAY[0..99] OF MG.Point;
    hotSpot: MG.Point;
    ..
MG.CreateSymbols(symbs,
    MG.outlineSymbols,
    FALSE,
    0,0,MG.nullPoint);
poly[0].x:= 10; poly[0].y:= 15;
poly[1].x:= 10; poly[1].y:= 18;
    ..
poly[29].x:= 10; poly[29].y:= 15;
hotSpot.x:= 10; hotSpot.y:= 15;
MG.AddSymbol(symbs,
            100,
            35,40,
            hotSpot,
            30*SizeOf(MG.Point),poly);
    ..
MG.SaveSymbols(symbs,'POINTERS.SOL');
MG.DisposeSymbols(symbs);

## LoadSymbols

P LoadSymbols(VAR symbs : Symbols; (*O*)
    filename : String;
    adjustAspect: BOOLEAN);

This procedure loads the symbol library in the file filename and attaches it to the symbol variable symbs.

If adjustAspect is TRUE and the symbol type is strokeSymbols, outlineSymbols or bezierSymbols, the symbols are scaled vertically with regard to the current device's aspect ratio. The symbols in the symbol file are assumed to have an aspect ratio of 1:1 (1000). If adjustAspect is FALSE or the symbol is of the type bitmapSymbols or deviceSymbols, no scaling is carried out.
If the size of the symbol library $\leq \text{maxRamSymbolsSize}$ (see "Variables" above), the symbols are loaded onto the ordinary program heap.

If the size of the symbol library $> \text{maxRamSymbolsSize}$, the symbols are instead loaded into the current virtual memory (real mode: conventional, extended XMS, expanded EMS or disk). See the descriptions of the unit VM, etc..

In order to make the loaded symbols the current symbols, the procedure SetSymbols must also be called. Windows, screens and devices can share the same symbol library and symbol variable.

\text{SV.Result}
\text{processWithoutWin} \quad \text{the calling process has no window}
\text{invalidSymbolsFile} \quad \text{invalid symbol file}
\text{outOfMem} \quad \text{not enough memory to load the library}
< 0 \quad \text{I/O error (the file is missing, invalid path, etc)}

Errors can also be reported from the virtual memory handler in VM.

\text{VAR}
\quad \text{syms: MG.Symbols;}
\quad ..
\quad \text{MG.LoadSymbols(syms,'MARKERS.SOS',TRUE);}
\quad \text{IF SV.Result <> SV.ok THEN ..}
\quad \text{MG.SetSymbols(syms);}
\quad \text{MG.SetPos(100,100);}
\quad \text{MG.DrawSymbol(75);}
\quad ..

\text{SaveSymbols}

\text{Procedure SaveSymbols(syms : Symbols;}
\quad \text{filename: String);}\
\text{Saves the symbol library attached to syms in a symbol file on disk with name and path in filename.}

This procedure is very useful when you have created your own symbols, modified old symbols or converted stroke, outline or Bézier symbols to bitmapped symbols and want to store them permanently.
SV.Result
processWithoutWin  the calling process has no window
invalidSymbols    invalid symbol file
< 0               I/O error (file not found, invalid path, etc)

Errors can also be reported from the virtual memory handler in VM.

DisposeSymbols

P DisposeSymbols(VAR symbs: Symbols (*IO*));

Erases a symbol library from memory (conventional or virtual).

N.B. that all references to symbs are rendered invalid from now on. This applies especially to symbol references in the various windows.

symbs is set to nullSymbols after the call is made.

SV.Result
processWithoutWin  the calling process has no window
invalidSymbols    invalid symbol variable

Errors can also be reported from the virtual memory handler in VM.

Symbol Library Information

GetSymbolsInfo

P GetSymbolsInfo(    symbs: Symbols;
                  VAR info : SymbolsInfo (*O*));

Reads and returns in info information about the symbol library symbs. The information is contained in a record variable of type SymbolsInfo (see "Data Types" above).

SV.Result
processWithoutWin  the calling process has no window
invalidSymbols    the current symbol library is missing or is invalid
If an error occurs, info will contain "garbage"! See also GetCurSymbols-Info.

Converting Symbols

**CreateBitmapSymbols**

```pascal
P CreateBitmapSymbols(    vectorSymbols: Symbols;
symbSet      : SymbolsSet;
scaleX,scaleY: Integer32;
pat          : Pattern;
style        : LineStyle;
lineWidth    : INTEGER;
VAR bitmapSymbols: Symbols (*O*));
```

Creates a new bitmapped symbol library, bitmapSymbols, from the stroke, outline or Bézier symbols in vectorSymbols. The new symbols are scaled using scaleX and scaleY (1000 = 1:1). If the original is an outline or Bézier symbol type, the new symbols will be filled using the pattern pat. Strokes, outlines and Béziers are drawn using the line style style and width lineWidth. Only those symbols whose numbers are included in the set symbSet and which at the same time are included in the original symbols, will be included in the bitmapSymbols.

If the original symbols are of type strokeSymbols the pattern pat is ignored. If the pattern pat equals nullPattern no filling will take place. If the line style style equals nullLineStyle no contour line will be drawn.

The new symbols library is not influenced by the current window's symbol transformation, but by any one-time transformation of vectorSymbols. Only scaling is allowed, rotation or shearing are not allowed.

The original symbols in vectorSymbols are not modified.

**SV.Result**

- invalidSymbols: invalid vector symbols
- invalidTransform: invalid transformation
- outOfMem: not enough memory to fulfil the conversion

Errors can also be reported from the virtual memory handler in VM.
MG.CreateBitmapSymbols(syms0,
[1..10],
500,500,
MGPat.GenPattern(MGPat.solidPat),
MG.nullLineStyle,
1,
syms1);
MG.SaveSymbols(syms1,'C:\symbols\SYMBOLS1.SBS');

Transformation of Symbol Libraries

TransformSymbols

P TransformSymbols(symbols : Symbols; (*Io*)
  scaleX,scaleY: Integer32;
  angle : INTEGER;
  shearX,shearY : INTEGER);

The symbols in the symbol library attached to symbols are transformed globally, i.e. all output devices, screens and windows are affected. The symbol library in symbols is modified permanently (i.e. until the original library is loaded from disk again or a new transformation is applied).

The symbols must have stroke, outline or Bézier type.

Global transformations are carried out in the following order:

1 Scaling takes place using the factor scaleX in the x direction and scaleY in the y direction. 1000 corresponds to 1:1, i.e. the size is unchanged.

2 Rotation takes place using the angle in the parameter angle (anticlockwise) which must be within the range of 0..3599 (decidegrees).

3 Shearing takes place using the factor shearX in the x direction and shearY in the y direction. The value 0 corresponds to no shearing, while 1000 corresponds to 1 unit.
SV.Result
processWithoutWin the calling process has no window
invalidSymbols invalid symbol variable
invalidSymbolsType only stroke, outline and Bézier symbols
invalidTransform invalid transformation

Errors are also reported from the virtual memory handler in VM.

VAR
  syms: MG.Symbols;
 ..
  MG.LoadSymbols(syms,'ELECTRIC.SSL',TRUE);
  MG.TransformSymbols(syms,1500,1500,900,0,0);

Editing Symbol Libraries

**GetSymbolData**

```
P GetSymbolData(    symbs     : Symbols;
                    symbol    : Word;
                    VAR symWidth,
                    symHeight : INTEGER;   (*O*)
                    VAR symHotSpot: Point;  (*O*)
                    VAR dataSize : Word;    (*IO*)
                    VAR data     : *O*));
```

This procedure returns information on the symbol **symbol** in the symbol library **symbs**. The width of the symbol is returned in **symWidth**, the height in **symHeight** and the location of the hot spot in **symHotSpot**. The "drawing" data of the symbol is copied to the buffer **data** and the size of this buffer is specified in in **dataSize** (normally **SizeOf(data)**) before the call is made. On return the **dataSize** parameter contains the real size of symbol data copied to **data**.

**data** must be large enough to accommodate the symbol data. Otherwise you will get the error **MGErr.outOfMem**.

The structure of **data** depends on the type of symbols:

If **symbs** is attached to bitmapped symbols, then **data** contains a bit block ((**symWidth** + 7) DIV 8 bytes per row).
If `symbs` is attached to stroke or outline symbols, then `data` is an array of points of the type `MG.Point`, if `symbs` contains "large" symbols (the number of symbols is `dataSize DIV SizeOf(MG.Point)`) or the type `MG.ShortPoint`, if `symbs` contains "small" symbols (the number of symbols is `dataSize DIV SizeOf(MG.ShortPoint)`).

If `symbs` is attached to Bézier symbols, then `data` is an array of Bézier elements of the type `MG.Bezier` (the number of symbols is `dataSize DIV SizeOf(MG.Bezier)`).

Data from `GetSymbolData` can be used as input to `AddSymbol`.

`SV.Result`  
`MGErr.invalidSymbols`  `MG.deviceSymbols` not allowed  
`MGErr.symbolOutOfRange`  `symbol` is not in the library  
`MGErr.invalidSymbolsType`  `MG.deviceSymbols` are not allowed  
`MGErr.outOfMem`  `data` can't hold the data

Errors are also reported from the virtual memory handler in `VM`.

```pascal
VAR  
  outlineSymbs,  
  strokeSymbs : MG.Symbols;  
  symWidth,  
  symHeight : INTEGER;  
  symHotSpot : MG.Point;  
  polyData : ARRAY[0..499] OF MG.Point;  
  polySize : Word;  
  sym : Word;  
  ..
MG.SetSymbols(outlineSymbs);  
FOR sym:= MG.CurFirstSymbol TO  
  MG.CurLastSymbol DO BEGIN  
  polySize:= SizeOf(polyData);  
  MG.GetSymbolData(outlineSymbs,  
    sym,  
    symWidth,symHeight,  
    symHotSpot,  
    polySize,polyData);  
  IF SV.Result = SV.ok THEN BEGIN  
    MG.ClrWin; (* Draw the symbol polygon on screen *)  
    MG.FillPolygon(polyData,  
      0,polySize DIV SizeOf(MG.Point) - 1);  
  END;
```
In the example above an outline symbol library is converted to a stroke library. The symbols are printed using their polygon descriptions, which we get from \texttt{GetSymbolData}.

\begin{quote}
\textbf{GetSymbolDataPtr}
\end{quote}

\begin{verbatim}
P GetSymbolDataPtr(  symbs     : Symbols;  
                     symbol    : Word;  
                     VAR symWidth,  
                     symHeight : INTEGER;  (*O*)  
                     VAR symHotSpot: Point;  (*O*)  
                     VAR dataSize : Word;  (*O*)  
                     VAR dataPtr  : Pointer  (*O*));
\end{verbatim}

This procedure returns information on the symbol \texttt{symbol} in the symbol library \texttt{symbs}. The width of the symbol is returned in \texttt{symWidth}, the height in \texttt{symHeight} and the location of the hot spot in \texttt{symHotSpot}. The "drawing" data of the symbol is copied to the pointer \texttt{dataPtr}, which is automatically allocated during the call (in contrast to \texttt{GetSymbolData}). The \texttt{dataSize} parameter contains the size of the symbol data copied to \texttt{dataPtr}.

\texttt{dataPtr} is allocated on each call to \texttt{GetSymbolDataPtr} and must therefore be deallocated when it is not needed any more, or before \texttt{GetSymbolDataPtr} is called again.

\texttt{dataPtr} is deallocated by the call \texttt{FreeMem(dataPtr, dataSize)}.

The structure of the data pointed to by \texttt{dataPtr} depends on the type of symbols:

If \texttt{symbs} is attached to bitmapped symbols, then \texttt{dataPtr} points to a bit block \((\texttt{symWidth} + 7) \div 8\) bytes per row.
If `symbs` is attached to stroke or outline symbols, then `dataPtr` points to an array of points of the type `MG.Point`, if `symbs` contains "large" symbols (the number of symbols is `dataSize` DIV `SizeOf(MG.Point)` or the type `MG.ShortPoint`, if `symbs` contains "small" symbols (the number of symbols is `dataSize` DIV `SizeOf(MG.ShortPoint)`).

If `symbs` is attached to Bézier symbols, then `dataPtr` points to an array of Bézier elements of the type `MG.Bezier` (the number of symbols is `dataSize` DIV `SizeOf(MG.Bezier)`).

Data from `GetSymbolDataPtr` can be used as input to `AddSymbol`.

**SV.Result**

- `MGErr.invalidSymbols`: `MG.devicesymbols` not allowed
- `MGErr.symbolOutOfRange`: symbol is not in the library
- `MGErr.invalidSymbolsType`: `MG.devicesymbols` are not allowed
- `MGErr.outOfMem`: not enough memory to allocate `dataPtr`

Errors are also reported from the virtual memory handler in VM.

```pascal
TYPE
  PointArray = ARRAY[0..999] OF MG.Point;
VAR
  outlineSymbs, strokeSymbs : MG.Symbols;
  symWidth, symHeight : INTEGER;
  symHotSpot : MG.Point;
  polyDataPtr : ^PointArray;
  polySize : Word;
  sym : Word;
  ..
MG.SetSymbols(outlineSymbs);
FOR sym:= MG.CurFirstSymbol TO MG.CurLastSymbol DO BEGIN
  MG.GetSymbolDataPtr(outlineSymbs,
    sym,
    symWidth,symHeight,
    symHotSpot,
    polySize,polyDataPtr);
  IF SV.Result = SV.ok THEN BEGIN
    MG.ClrWin; (* Draw the symbol polygon on screen *)
    MG.FillPolygon(polyData^,
      0,polySize DIV `SizeOf(MG.Point)` - 1);
  END;
```

MultiGraphics & Borland Pascal/Turbo Pascal  DATABITEN
(* Add to the stroke symbol library *)
MG.AddSymbol(strokeSymbs,
    sym,
    symWidth,symHeight,
    symHotSpot,
    polySize,polyDataPtr^);
    FreeMem(polyDataPtr,polySize);
END;
END; (*for*)
..

This is the same example as listed under GetSymbolData.

### DeleteSymbol

```pascal
P DeleteSymbol(syms : Symbols; (*Io*)
    symbol: Word);
```

Removes the symbol `symbol` from the symbol library attached to `symbs`. Symbol data is lost.

**SV.Result**

- MGErr.invalidSymbols: `symbol` is not found
- MGErr.symbolOutOfRange: symbol is not found
- MGErr.outOfMem: out of memory for temporary buffer (needed when `symbs` is in virtual memory)

Errors are also reported from the virtual memory handler in VM.

### AddSymbol

```pascal
P AddSymbol(
    symbs : Symbols; (*Io*)
    symbol : Word;
    symWidth,
    symHeight : INTEGER;
    symHotSpot: Point;
    dataSize : Word;
    VAR data (*I*)
);
```

Expands the symbol library attached to `symbs` with data for the symbol `symbol`. `symWidth`, `symHeight` and `symHotSpot` contain the width, height and hot spot location of the symbol. The symbol description data is given in `data` and encompasses `dataSize` bytes.
If there is already a symbol with number symbol, this will be replaced by the new symbol. If the symbol library has a fixed width, a fixed height and/or a fixed hot spot location, the data in symWidth, symHeight and symHotSpot must correspond to these fixed values.

The structure of data depends on the type of symbol library:

If symbs is attached to bitmapped symbols, then data is a bit block ((symWidth + 7) DIV 8 bytes per row).

If symbs is attached to stroke or outline symbols, then data is an array of points of the type MG.Point, if symbs contains "large" symbols (dataSize = <number of points> * SizeOf(MG.Point)) or the type MG.ShortPoint, if symbs contains "small" symbols (dataSize = <number of points> * SizeOf(MG.ShortPoint)).

If symbs is attached to Bézier symbols, then data is an array of Bézier elements of the type MG.Bezier (dataSize = <number or elements> * SizeOf(MG.Bezier)).

When calling AddSymbol the internal offset table is expanded to its maximum size, if this has not been done before. If the library is saved on disk, using SaveSymbols, the offset table is minimized. Therefore, when the library is reloaded from disk using LoadSymbols, the library will usually occupy less heap space than before.

Output data from GetSymbolData or GetSymbolDataPtr can be used as input data to AddSymbol.

SV.Result
MGErr.invalidSymbols MG.deviceSymbols not allowed
MGErr.symbolOutOfRange symbol is not in the library
MGErr.invalidSymbolWidth invalid symbol width (< 0, > 127 if "small" symbols, or different from a fixed width)
MGErr.invalidSymbolHeight invalid symbol height (< 0, > 127 if "small" symbols, or different from a fixed height)
MultiGraphics & Borland Pascal/Turbo Pascal

**MGErr.invalidHotSpot**
invalid hot spot location (< -127 or > 127 if "small" symbols, or different from a fixed hot spot)

**MGErr.invalidSymbolData**
invalid **dataSize** (too large or not an integer number of points or Bézier elements)

**MGErr.symbolTooLarge**
**dataSize** too large

**MGErr.outOfMem**
out of memory for temporary buffer (if **symb**s use virtual memory) or for offset table

Errors are also reported from the virtual memory handler in **VM**.

```
CONST
  minInt = -32768;

TYPE
  PolyArray12 = ARRAY[0..11] OF MG.Point;

CONST
  pa1: PolyArray12 = (  
    (x: 0;y: 0),        
    (x:30;y: 0),        
    (x:30;y:30),        
    (x: 0;y:30),        
    (x: 0;y: 0),        
    (x:minInt;y:minInt),
    (x: 5;y: 5),        
    (x:25;y: 5),        
    (x:25;y:25),        
    (x: 5;y:25),        
    (x: 5;y: 5),        
    (x:minInt;y:minInt));

VAR
  symb   : MG.Symbols;
  hotSpot: MG.Point;
  ...
  MG.CreateSymbols(symb, MG.outlineSymbols, FALSE,  
                   0,0,MG.nullPoint);  
  hotSpot.x:= 15; hotSpot.y:= 15;  
  MG.AddSymbol(symb,1,31,31,hotSpot, SizeOf(pa1),pa1);  
  MG.SetSymbols(symb);  
  MG.SetPos(100,100); MG.DrawSymbol(1);
  ...
  MG.SaveSymbols(symb, 'TEST.SOL');  
  MG.DisposeSymbols(symb);
```
Current Symbol Library

**SetSymbols**

```pascal
P SetSymbols(syms: Symbols);
```

Sets the current symbols in the current window. Prior to this `syms` must have been attached to a symbol library using `LoadSymbols`, `CreateSymbols` or `CreateBitmapSymbols`.

The window oriented symbol transformation "is reset", i.e. the scale factor is set to 1000 (1:1), the rotation angle to 0, and the shearing factors to 0.

If you do not intend to use a symbol library in the window, `SetSymbols` should be called using the value `nullSymbols`.

When a window is created, the current symbol library is set to the default symbols of the device.

**SV.Result**

- `processWithoutWin` the calling process has no window
- `invalidSymbols` invalid symbol variable

```pascal
VAR
    win : MG.Window;
    syms: MG.Symbols;
    ..
    MG.SetWin(win);
    MG.SetSymbols(syms);
```

**SetDefaultSymbols**

```pascal
P SetDefaultSymbols;
```

Sets the device's *default symbols* as the current symbols in the current window. The default symbols have no open symbols variable and have therefore their own user procedure in order to be activated.

When a window is created, the default symbols are assigned as the current symbols.

**SV.Result**

- `processWithoutWin` the calling process has no window
VAR
    syms: MG.Symbols;
    ..
    MG.SetSymbols(syms);
    MG.DrawSymbol(6);
    MG.SetDefaultSymbols;

CurSymbols

F CurSymbols: Symbols;

Returns a *reference* to the current symbol library in the current window. If
the symbol library is missing, **nullSymbols** is returned.

SV.Result
processWithoutWin  the calling process has no window

If an error occurs, **nullSymbols** is returned.

CurSymbolsType

F CurSymbolsType: SymbolsType;

This functions returns the symbol type of the current symbol library in the
current window.

SV.Result
processWithoutWin  the calling process has no window
invalidSymbols    no symbol library installed or damaged

If an error occurs, **maxSymbolsType** is returned.

IF MG.CurSymbolsType <> MG.bitmapSymbols THEN
    MG.TransformSymbols(sym,2000,2500,0,0,0);
    ..
**CurSymbolWidth**

```pascal
F CurSymbolWidth(symbol: Word): INTEGER;
```

Returns the width of a symbol with the number `symbol` in the current window's symbol library. Returns 0 if the symbol is missing.

**SV.Result**

- `processWithoutWin`: the calling process has no window
- `invalidSymbols`: the current symbol library is missing or is invalid

If an error occurs, `MAXINT` is returned.

```pascal
MG.DrawSymbol(75);
MG.SetRelPos(MG.CurSymbolWidth(75),0);
```

**CurSymbolHeight**

```pascal
F CurSymbolHeight(symbol: Word): INTEGER;
```

Returns the height of a symbol with the number `symbol` in the current window's symbol library.

**SV.Result**

- `processWithoutWin`: the calling process has no window
- `invalidSymbols`: the current symbol library is missing or is invalid
- `symbolOutOfRange`: invalid `symbol` value

If an error occurs, `MAXINT` is returned.

**GetHotSpot**

```pascal
P GetHotSpot(    symbol: Word;
    VAR hot   : Point (*O*));
```

Returns the hot spot location of a symbol with the number `symbol` in the current window's symbol library. The hot spot is the point around which the symbol is drawn. The variable parameter `hot` contains the x and y value for the hot spot.

**SV.Result**
**processWithoutWin**  
the calling process has no window

**invalidSymbols**  
the current symbol library is missing or is invalid

**symbolOutOfRange**  
invalid symbol value

```pascal
VAR
  hot: MG.Point;
.. MG.GetHotSpot(75,hot);
IF hot.x > 10 THEN ..
```

### CurFirstSymbol

**F CurFirstSymbol: Word;**

Returns the number of the first symbol in the current symbol library and window.

**SV.Result**

- **processWithoutWin**  
  the calling process has no window

- **invalidSymbols**  
  the current symbol library is missing or is invalid

If an error occurs, **65535** is returned.

```pascal
FOR symbol:= MG.CurFirstSymbol TO MG.CurLastSymbol DO ..
```

### CurLastSymbol

**F CurLastSymbol: Word;**

Returns the number of the last symbol in the current symbol library and window.

**SV.Result**

- **processWithoutWin**  
  the calling process has no window

- **invalidSymbols**  
  the current symbol library is missing or is invalid

If an error occurs, **65535** is returned.
GetCurSymbolsInfo

P GetCurSymbolsInfo(VAR info: SymbolsInfo (*O*));

Reads and returns in info information about the current symbol library in the current window. The information is contained in a record of the type SymbolsInfo (see "Data Types" above).

SV.Result
processWithoutWin the calling process has no window
invalidSymbols the current symbol library is missing or is invalid

If an error occurs, info will contain "garbage"! See also GetSymbolsInfo.

Window Transformation of Symbols

SetSymbolsTransform

P SetSymbolsTransform(scaleX, scaleY: Integer32;
angle : INTEGER;
shearX, shearY: INTEGER);;

Sets the current local window transformation for symbols. When a symbol is drawn, it is first transformed using this window specific transformation. The symbols must be of the stroke, outline or Bézier type.

scaleX and scaleY specify the scale factors for the x and y directions. The value 1000 corresponds to 1:1.

angle specifies the rotation angle in tenths of degrees (decidegrees).

shearX and shearY specify the shearing factor in the x and y directions. The value 0 gives no shearing. The value 1000 corresponds to shearing factor 1.

When a window is created, the scale factor is assigned to 1000, the rotation angle to 0 and the shearing factors to 0. These values also give the fastest drawing speed.

Each window can have its own symbol transformation. This does not prevent different window sharing the same symbol library.
Ref 14-32

SV.Result
processWithoutWin  the calling process has no window
invalidSymbols  the current symbol library is missing or
                is invalid
invalidAngle  invalid rotation angle
invalidSymbolsType  bitmapped symbols cannot be
                transformed
invalidTransform  invalid transformation

MG.SetWin(win1);
MG.SetSymbolsTransform(1000,1500,0,500,0);
MG.SetPos(100,100); MG.DrawSymbol(123);
MG.SetSymbolsTransform(1000,1500,450,0,0);
MG.SetPos(150,100); MG.DrawSymbol(124);

GetSymbolsTransform

P GetSymbolsTransform(VAR scaleX, scaleY: Integer32;(*O*)
VAR angle        : INTEGER;(*O*)
VAR shearX,shearY: INTEGER (*O*));

Reads the current local symbol transformation in the current window.

SV.Result
processWithoutWin  the calling process has no window

Draw Symbol

DrawSymbol

P DrawSymbol(symbol: Word);

Draws a symbol with the number symbol found in the current symbol
library and current window. The symbol is drawn with its hot spot in the
current position and using the current window mode, i.e. it is transformed
using the current symbol transformation (if it is of the stroke, outline or
Bézier type). It is clipped against the current viewport and is drawn using
the current foreground and background color, color mask and drawing
mode.

SV.Result
processWithoutWin  the calling process has no window
invalidSymbols  the current symbol library is missing or is invalid
symbolToComplex stroke or outline symbols use too many points
deviceProcNotInUse filling in of outline symbols not possible
symbolOutOfRange invalid symbol value

Errors can also be reported from the virtual memory handler in VM and from each respective device driver.

MG.SetPos(200,200); MG.DrawSymbol(150);
MG.SetPos(100,100); MG.DrawSymbol(160);
MultiGraphics supports text output in graphics modes. Graphics text allows you to work with a greater variety of fonts and sizes.

In the unit MG, you will find all the basic data types and routines needed for font and text handling. The MGIO unit contains more advanced routines for text based I/O. Finally, MGFnt contains some procedures for font conversions.

A font is made up of individual characters with a particular style and size. Each font can store up to 256 characters. Normally, the international standard characters in accordance with ASCII/ISO are used. In the PC DOS environment, IBM's extended 8 bit ASCII code is often used.

Each device can have its own default font. Other fonts can be created or copied and modified from other fonts and eventually stored in font files. Fonts from font files can be loaded again. Active fonts are always linked to variables of the type Font.

Each window has its own current font. Fonts can be shared by several windows.
Fonts are always *monochrome* and therefore are drawn using the current foreground and background colors. **MultiGraphics** has five different *font types*, namely:

- bitmapped fonts (*bitmapSymbols*),
- stroke fonts (*strokeSymbols*),
- outline fonts (*outlineSymbols*),
- Bézier fonts (*bezierSymbols*),
- device specific fonts (*deviceSymbols*).

Stroke, outline and Bézier fonts can be *globally transformed*, i.e. be scaled, rotated and sheared, in a way that is not dependent on the window. All output devices, screens and windows that use the same font variable are affected by the transformation.

Shearing entails a deformation of the characters in the x and y directions. Italicized text is an example of a font that has been "sheared".

Fonts are related to symbols; this relationship is described later.

### Global Resources

Fonts are *global resources*. This means that they are created, loaded and deleted independently of the active devices and may be shared by several devices, screens and windows. This also means that fonts must be deallocated manually from memory by calling **MG.DisposeFont**. They are not disposed of automatically when closing device drivers.

An exception from this is the built-in default font in the device driver. This is removed from memory together with the driver.

### The Current Font

Each window has its own *current font*. When a window is created, the device's *default font* is activated automatically. This font is unique for the current device. The font is normally enclosed in the device driver.
Font Types

MultiGraphics accepts five different font types:

![Bitmapped, stroke, outline and bezier fonts](image)

**Fig 1: Bitmapped, stroke, outline and bezier fonts**

**Bitmapped Fonts**

Bitmapped fonts are made up of bit block patterns. The main advantages are:

+ high speed,
+ exact control of the appearance of characters.

Bitmapped fonts can often be drawn at high speeds, especially if the output is restricted to "speed positions" (see `GetFastXY` in MG). Stroke, outline and Bézier fonts can be converted to bitmapped fonts.

The disadvantages of bitmapped fonts are:

- cannot be scaled, rotated or sheared,
- can only be used with raster devices,
- take up a lot of memory when large characters are being used,
- drawing speed is reduced by a power of two in proportion to the character dimensions.
Stroke Fonts

Stroke fonts are made up of straight line segments that form one or more polylines for each character. The advantages of this type of font are that the characters:

+ can be scaled, rotated and sheared,
+ work on all types of devices,
+ memory requirement is not dependent on character size,
+ drawing speed is reduced linearly compared to the character dimensions.

The disadvantages are:

- they are usually slower than small bitmapped fonts,
- they may look "sparse" because the characters are constructed with lines,
- they need lots of points to create a smooth appearance,
- they appear jagged when reduced,
- they appear unsmoothed when enlarged too much.

Outline Fonts

Outline fonts are constructed with line segments, in the same way as stroke fonts. In contrast to stroke fonts, however, line segments always form closed polygons, which are filled using the current pattern. If the current pattern is nullPattern, filling is not carried out. The outline lines are drawn at the same time as the characters. If the current line style is nullLineStyle the outline is not drawn. The advantages of outline fonts are:

+ can be scaled, rotated and sheared,
+ can be filled with patterns which means that characters fill up the drawing area better,
+ can be used on all types of devices that support polygon filling.

The disadvantages include:

- they are slower than stroke fonts and bitmapped fonts,
- they require support for polygon filling,
- they need lots of points to create a smooth appearance,
- they appear jagged when reduced,
- they appear unsmoothed when enlarged too much.
Bézier Fonts

Bézier fonts are constructed with Bézier curve elements. Bézier curves usually belong to a class of third degree polynomials with good properties that make them suitable for the description of symbols and characters. In contrast to straight edges of outline characters, points in Bézier outlines can be computed to any level of precision, which makes the appearance smooth.

Bézier characters always form closed curves, which are filled using the current pattern. If the current pattern is `nullPattern`, filling is not carried out. The outline curve is drawn at the same time as the character. If the current line style is `nullLineStyle` the outline is not drawn. The advantages of Bézier fonts are:

+ can be scaled, rotated and sheared and still appear smooth,
+ can be filled with patterns,
+ can be used on all types of devices that support polygon filling.

The disadvantages include:

- they are slower than outline, stroke and bitmapped fonts,
- they require support for polygon filling,
- they appear jagged when reduced too much.

Device Specific Fonts

Many devices, such as dot matrix printers and laser printers, have one or more in-built fonts. The advantages of these device specific fonts are:

+ they are tailor-made for the hardware,
+ they are usually of good quality,
+ they do not take so much space in memory.

The disadvantages include:

- only work with one particular device
- certain drivers can only handle a limited number of text strings for device specific fonts.

Converting to Bitmapped Fonts

You may convert scaled stroke, outline and Bézier fonts to bitmapped fonts.
by calling `CreateBitmapFont`. The converted fonts can then be saved in a
font file by calling `SaveFont`. This approach combines the best of two
worlds, namely the speed of bitmapped fonts and the flexibility of vector
and Bézier fonts when scaling and doing patterns.

Font and Character Information

All characters in a particular font have the same *height* and the same height
of the *base line*. The characters are drawn in line with the base line, i.e. the
characters are placed so that the base line is in line with the current y posi-
tion.

![Character Information](image)

**Fig 2: Character information**

A font can consist of characters with *fixed widths* or characters with *varying
widths*, often called *proportional fonts*.

The values of the width, height and base line are modified when the font is
transformed.

Large and Small Fonts

*MultiGraphics* supports two different maximum sizes of characters, namely:

- small characters with a maximum width and height of <= 127 pixels,
- large characters with a maximum width and height of <= 32767 pixels.

These limits must not be exceeded when doing font transformations (stroke,
outline and Bézier fonts).
"Small" fonts require less memory, while "large" fonts allow greater choice when scaling. When doing printouts on devices with high resolutions, such as laser printers and pen plotters, it may be necessary to use "large" fonts.

Bézier fonts are always of "large" character sizes.

Each font format has its own file format.

Font Files

*MultiGraphics* has its own unique format for font files. A font file can store up to 256 characters. All characters in a font file are of the same type (bitmapped, stroke, outline, Bézier or device specific). Besides the description of the individual characters, the font file also contains common information about the characters' height, baseline, maximum character width, average width etc., as well as specific information about the width of individual characters.

We recommend using the following file extensions for font files:

- **CBL**  
  "Characters Bitmapped Large"
- **CBS**  
  "Characters Bitmapped Small"
- **CSL**  
  "Characters Stroke Large"
- **CSS**  
  "Characters Stroke Small"
- **COL**  
  "Characters Outline Large"
- **COS**  
  "Characters Outline Small"
- **CZL**  
  "Characters BéZier Large"
- **CD**  
  "Characters Device"

Creating and Modifying Fonts

In *MG* you will find procedures for creating new fonts, modifying old fonts and saving the fonts on disk. Character data can be read from existing fonts.

A new and empty font is created by calling *CreateFont*. The new font is of a specific type (bitmapped, stroke, outline or Bézier) and size ("small" or "large") and is moreover characterized of having fixed or non-fixed character widths and a fixed character height and fixed base line.

Bézier fonts can only have "large" characters.

You can add new characters, or change old characters in a font by calling
AddChar. The character data is dependent on the type of font as follows:

Bitmapped characters are described by a bit block (see "MG - Copy Graphics"), where the bits set to "1" are used for drawing pixels and bits set to "0" for not drawing.

Stroke and outline characters are described by an array of points. "Small" fonts use points of the type MG.ShortPoint, while "large" fonts use points of MG.Point type. MG.nullPoint and MG.nullShortPoint are used to create "jumps" in the curve (see DrawPolyline and DrawShortPolyline in "MG - Lines" and FillPolygon and FillShortPolygon in "MG - Patterns").

Bézier characters are described by an array of Bézier elements of the type MG.Bézier, elements that in turn are defined by an array of four control points of MG.Point type. The curve is continuous if the endpoint of one Bézier element is the starting point of the next element. Otherwise there will be a "jump" in the curve (see DrawPolyBezier in "MG - Lines" and FillPolyBezier in "MG - Patterns").

You can get all information on a character, in an existing font, by calling GetCharData or GetCharDataPtr. This will return information on the current character width, the fixed characters heights, base line and character data (the structure is dependent on the type of font). The structure of the information received is the same as the structure of the information fed to AddChar, so that you can use the information returned as input to AddChar.

The difference between GetCharData and GetCharDataPtr is that the first procedure puts data in an existing buffer supplied by the caller, while the second procedure allocates a new data buffer on the program heap and returns a pointer to the data.

Characters can be deleted from a font by calling DeleteChar. Fonts can be stored permanently in font files on disk, by calling SaveFont. They can be loaded by calling LoadFont (you can also load font files as symbols using LoadSymbols).

Taken as a whole the procedures in this overview can be used to create completely new fonts from scratch, they can be used to modify and cut from old fonts or to borrow characters from one library to another. The result can be stored in font files on disk.
Transformations

Fonts of the stroke, device or Bézier specific types can be scaled, rotated and sheared.

*Scaling* means that something is changed in size. When scaling is carried out, the x and y values are multiplied by scale factors. The x and y coordinates can have different scale factors. Scale factors are specified in units of parts per thousand, which means that the value of 1000 corresponds to a scale of 1:1, i.e. no change in size. The scale factor of 2000 means doubling the scale, and so on. To avoid rounding errors it is best to use "integer" scale factors, i.e. 2000, 3000, 4000, etc.. If the scale factor < 1000 it is impossible to avoid rounding errors.

*Rotation* is specified by an angle given in tenths of degrees (decidegrees). 10 decidegrees correspond to 1 degree. 3600 decidegrees correspond to 360 degrees, i.e. one complete revolution. Positive angles give an anti-clockwise rotation while negative angles rotate clockwise.

*Shearing* means that characters are deformed ("slanted") in the x or y direction. The x and y directions can have different shearing factors. Shearing is specified in units of parts per thousand, which means that a value of 1000 corresponds to a factor of 1. The value 0 entails that no shearing occurs at all. When shearing takes place in the x direction, the characters are "italicized".

Fonts are always transformed *globally*, i.e. independent of output device, screen and window, and they are transformed once and for all, because the complete description of the font attached to a variable of the `Font` type is transformed. No time consuming calculations need therefore be carried out before the characters are printed out.

A transformed font can be transformed yet again, but repeated transformations can lead to rounding off errors. If you need to use the same font with different transformations it is better to load the font file again, or load the font file several times and attach it to different `Font` variables.
Fonts and Symbols

Fonts and symbols are closely related. In fact, each font file, except device specific font files (of the type deviceSymbols), can also be loaded and used in the same way as symbol files. The opposite, however, does not apply in every case - in general, a symbol file cannot be loaded in the same way as a font file.

Fonts and symbols use the same representations. These can be of four types, bitmapped (bitmapSymbols), stroke type (strokeSymbols), outline type (outlineSymbols) or Bézier type (bezierSymbols).

The only difference is the following: characters in a font can have different widths but must have the same height and base line. This restriction does not apply to symbols - individual symbols can have different heights and hot spots (a broader concept than the term base line).

A font file can therefore always be loaded in the same way as a symbol file (if it is not a device specific font).

Examples

GA12 and GAFNT display some of the standard fonts and font conversion. GAMkFont shows how to create a new font from an image with characters.

Data Types

SymbolsType  = (bitmapSymbols, strokeSymbols, bezierSymbols, outlineSymbols, deviceSymbols);

Symbol and font types in MultiGraphics. These types are described above.
Font

Data type for fonts. All fonts must be attached to variables of the type Font. There is but one exception; the default font has a hidden font variable.

Constants

nullFont

Defines a non-existing font.

Procedures & Functions

Creating, Loading and Saving Fonts

CreateFont

P CreateFont(VAR fnt : Font; (*O*)
    fontType : SymbolsType;
    shortFont : BOOLEAN;
    fixWidth : INTEGER;
    charHeight : INTEGER;
    charBaseline: INTEGER);

This procedure creates a new and empty font and attaches it to the variable fnt. The type of font is given in fontType (bitmapped, stroke, outline or Bézier type, not a device specific font). If shortFont is set to TRUE a font of "small" characters (max. 127x127 pixels) is created and FALSE creates a font of "large" characters.

fixWidth >= 0 creates a font of fixed width characters. fixWidth <= 0 creates a font where the width of the characters can vary.
**charHeight** > 0 specifies the fixed character height. **charBaseline** > 0 specifies the height of the font base line.

The new font is created in the current virtual memory and has an offset table of maximum length that can store up to 255 characters. New characters are added by calling **AddChar**.

Fonts can be saved in font files on disk by calling **SaveFont**. When the font is saved, the offset table is minimized. When the font is reloaded from disk using **LoadFont**, the font will therefore usually need less heap space than before.

**SV.Result**
**MGErr.outOfMem** not enough memory to create the font
**MGErr.invalidSymbols** **MG.deviceSymbols** not allowed

Errors can also be reported from the virtual memory handler in **VM**.

```pascal
VAR
  fnt : MG.Font;
  poly: ARRAY[0..99] OF MG.ShortPoint;
 ..
MG.CreateFont(fnt,
  MG.outlineSymbols,
  TRUE,
  0,40,15);
poly[0].x:= 10; poly[0].y:= 15;
poly[1].x:= 10; poly[1].y:= 18;
 ..
poly[29].x:= 10; poly[29].y:= 15;
MG.AddChar(fnt,
  100,
  35,
  30*SizeOf(MG.ShortPoint),poly);
 ..
MG.SaveFont(fnt,'MYCHARS.COL');
MG.DisposeFont(fnt);
```

**LoadFont**

```pascal
P LoadFont(VAR fnt : Font; (*O*)
  filename : String;
  adjustAspect: BOOLEAN);
```

This loads the font in the file **filename** and links it to the font variable **fnt**.
If `adjustAspect` is `TRUE` and the font type is `strokeSymbols`, `outline-Symbols` or `bezierSymbols`, the fonts are scaled vertically with regard to the current device's aspect ratio. The characters in the font file are assumed to have an aspect ratio of 1:1 (1000). If `adjustAspect` is `FALSE` or the font is of the type `bitmapSymbols` or `deviceSymbols`, no scaling is carried out.

If the font size <= `maxRamSymbolsSize` (see "Variables" in "MG - Symbols"), the font is loaded onto the ordinary program heap.

If the size of the font > `maxRamSymbolsSize`, the font is instead loaded into current virtual memory (real mode: conventional, extended XMS, expanded EMS or disk).

In order to activate a font, the procedure `SetFont` must also be called. Windows, screens and devices can share the same font.

```
SV.Result
processWithoutWin     the calling process has no window
invalidFontFile      invalid font file
outOfMem            not enough memory to load the font
< 0                  I/O error (the file is missing, invalid path, etc)
```

Errors can also be reported from the virtual memory handler in `VM`.

```
VAR
  fnt1,fnt2     : MG.Font;
  win1,win2,win3: MG.Window;
 ..
  MG.LoadFont(fnt1,'COURIER.CBS',FALSE);
  MG.LoadFont(fnt2,'ROMAN.CSS',TRUE);
  MG.SetWin(win1); MG.SetFont(fnt1);
  MG.SetWin(win2); MG.SetFont(fnt1);
  MG.SetWin(win3); MG.SetFont(fnt2);
```

Save the font attached to `fnt` in a font file on disk with name and path in `filename`.

```
SaveFont
P SaveFont(fnt     : Font;
           filename: String);
```

Saves the font attached to `fnt` in a font file on disk with name and path in `filename`. 
This procedure is very useful when you have created new fonts, modified old fonts or converted stroke, outline or Bézier fonts to bitmapped fonts and want to store these fonts permanently.

SV.Result
processWithoutWin the calling process has no window
invalidSymbols invalid font file
< 0 I/O error (file not found, invalid path, etc)

Errors can also be reported from the virtual memory handler in VM.

**DisposeFont**

```pascal
P DisposeFont(VAR fnt: Font (*IO*));
```

Erases a font from memory (conventional or virtual).

N.B. that all references to `fnt` are rendered invalid from now on. This applies especially to font references in the various windows.

`fnt` is set to `nullFont` after the call is made.

SV.Result
processWithoutWin the calling process has no window
invalidSymbols invalid font variable

Errors can also be reported from the virtual memory handler in VM.

```pascal
VAR
  fnt: MG.Font;
  ..
  MG.LoadFont(fnt,'HPHELV.CD',FALSE);
  ..
  MG.DisposeFont(fnt);
```
Font Information

**GetFontInfo**

\[
\text{P GetFontInfo}( \ fnt : \text{Font}; \\
\quad \text{VAR info: SymbolsInfo (*O*)}); \\
\]

Reads and returns in `info` information about the font `fnt`. The information is contained in a record in accordance with the type `SymbolsInfo` (see Ref "MG - Symbols" under "Data Types").

**SV.Result**

- `processWithoutWin` the calling process has no window
- `invalidSymbols` the current font is missing or is invalid

If an error occurs, `info` will contain "garbage"! See also `GetCurFontInfo`.

Converting Fonts

**CreateBitmapFont**

\[
\text{P CreateBitmapFont}( \ \\
\quad \text{vectorFont} : \text{Font}; \\
\quad \text{chars} : \text{CharSet}; \\
\quad \text{scaleX, scaleY} : \text{Integer32}; \\
\quad \text{pat} : \text{Pattern}; \\
\quad \text{style} : \text{LineStyle}; \\
\quad \text{lineWidth} : \text{INTEGER}; \\
\quad \text{VAR bitmapFont} : \text{Font (*O*)}); \\
\]

Creates a new bitmapped font, `bitmapFont`, from the stroke, outline or Bézier font in `vectorFont`. The new font is scaled using `scaleX` and `scaleY` (1000 = 1:1). If the original is an outline or Bézier font type, the new font will be filled using the pattern `pat`. Strokes, outlines and Bézier fonts are drawn using the line style `style` and width `lineWidth`. Only characters included in the set `chars`, which at the same time are included in the original font, will be included in the `bitmapFont`. 
If the original font is of type `strokeSymbols` the pattern `pat` is ignored. If the pattern `pat` equals `nullPattern` no filling will take place. If the line style `style` equals `nullLineStyle` no contour line will be drawn.

The new font is influenced by one-time transformations of `vectorFont`. Only scaling is allowed, rotation or shearing are not allowed.

The original font in `vectorFont` is not modified.

```pascal
MG.CreateBitmapFont(outlineFnt,
                      ['A'..'z','0'..'9'],
                      700,700,
                      MGPat.GenPattern(MGPat.halftone5),
                      MGPat.GenLineStyle(MGPat.solid),
                      1,
                      bitmapFnt);
```

Errors can also be reported from the virtual memory handler in VM.

`MG.TransformFont` is `Font`:

```pascal
P TransformFont(fnt : Font; (*Io*)
                scaleX, scaleY: Integer32;
                angle : INTEGER;
                shearX, shearY: INTEGER);
```

The characters in the font linked to `fnt` are `globally` transformed, i.e. the transformation affects all output devices, screens and windows. The font in `fnt` is modified permanently (i.e. until the original font is loaded from disk again or a new transformation is applied).

The font must have stroke, outline or Bézier format.
Global transformations are carried out in the following order:

1 *Scaling* takes place using the factor $\text{scaleX}$ in the x direction and $\text{scaleY}$ in the y direction. 1000 corresponds to 1:1, i.e. the size is unchanged.

2 *Rotation* takes place using the angle $\text{angle}$ (anti-clockwise), which must be within the range of 0..3599 (decidegrees).

3 *Shearing* takes place using the factor $\text{shearX}$ in the x direction and $\text{shearY}$ in the y direction. The value 0 corresponds to no shearing, while 1000 corresponds to 1 unit.

**SV.Result**

- **processWithoutWin**: the calling process has no window
- **invalidFont**: invalid font variable
- **invalidFontType**: only stroke, outline and Bézier fonts
- **invalidTransform**: invalid transformation

Errors are also reported from the virtual memory handler in VM.

```pascal
VAR
  fnt: MG.Font;
  ..
  MG.LoadFont(fnt,'BOLD.COL',FALSE);
  MG.\texttt{TransformFont}(fnt,3000,3000,0,500,0);
  MG.SetFont(fnt);
```

## Editing Fonts

### GetCharData

```pascal
P \texttt{GetCharData} ( fnt : Font; ch : CHAR;
  VAR charWidth : INTEGER; (*O*)
  VAR charHeight : INTEGER; (*O*)
  VAR charBaseline: INTEGER; (*O*)
  VAR dataSize : Word; (*IO*)
  VAR data : Word; (*O*)
)
```

This procedure returns information on the character $\text{ch}$ in the font $\text{fnt}$. The width of the character is returned in $\text{charWidth}$, the height in $\text{charHeight}$ and the height of the base line in $\text{charBaseline}$. The "drawing" data of the
character is copied to the buffer data and the size of this buffer is specified in dataSize (normally SizeOf(data)) before the call is made. On return the dataSize parameter contains the real size of the character data copied to data.

\textit{data must be large enough to accommodate the character data. Otherwise you will get the error MGErr.outOfMem.}

The structure of data depends on the character type:

If fnt is attached to bitmapped characters, then data contains a bit block ((charWidth + 7) DIV 8 bytes per row).

If fnt is attached to stroke or outline characters, then data is an array of points of the type MG.Point, if fnt contains "large" characters (the number of characters is dataSize DIV SizeOf(MG.Point)) or the type MG.ShortPoint, if fnt contains "small" characters (the number of characters is dataSize DIV SizeOf(MG.ShortPoint)).

If fnt is attached to Bézier characters, then data is an array of Bézier elements of the type MG.Bezier (the number of characters is dataSize DIV SizeOf(MG.Bezier)).

Data from GetCharData can be used as input to AddChar.

\textbf{SV.Result}

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGErr.invalidSymbols</td>
<td>MG.deviceSymbols not allowed</td>
</tr>
<tr>
<td>MGErr.symbolOutOfRange</td>
<td>ch is not in the font</td>
</tr>
<tr>
<td>MGErr.invalidSymbolsType</td>
<td>MG.deviceSymbols are not allowed</td>
</tr>
<tr>
<td>MGErr.outOfMem</td>
<td>data can't hold the data</td>
</tr>
</tbody>
</table>

Errors are also reported from the virtual memory handler in VM.

\textbf{VAR}

outlineFont, strokeFont : MG.Font; charWidth, charHeight : INTEGER; baseLine : MG.Point; polyData : ARRAY[0..499] OF MG.Point; polySize : Word; ch : CHAR;

MG.SetFont(outlineFont);
FOR ch:= MG.CurFirstChar TO
MG.CurLastChar DO BEGIN
polySize:= SizeOf(polyData);
MG.GetCharData(outlineFont,
    ch,
    charWidth,charHeight,
    baseline,
    polySize,polyData);
IF SV.Result = SV.ok THEN BEGIN
    MG.ClrWin; (* Draw the character polygon on screen *)
    MG.DrawPolyline(polyData,
          0,polySize DIV SizeOf(MG.Point) - 1);
    (* Add character to stroke font *)
    MG.AddChar(strokeFont,
        ch,
        charWidth
        polySize,polyData);
END;
END; (*for*)
.
.
In the example above an outline font is converted to a stroke font. The characters are printed using their polygon descriptions, which we get from GetCharData.

**GetCharDataPtr**

```pascal
P GetCharDataPtr(     fnt : Font;
    ch : CHAR;
    VAR charWidth : INTEGER
    VAR charHeight : INTEGER;(*O*)
    VAR charBaseline : INTEGER;(*O*)
    VAR dataSize : Word; (*O*)
    VAR dataPtr : Pointer (*O*));
```

This procedure returns information on the character ch in the font fnt. The width of the character is returned in charWidth, the height in charHeight and the height of the base line in charBaseline. The "drawing" data of the character is copied to the pointer dataPtr, which is automatically allocated during the call (in contrast to GetCharData). The dataSize parameter contains the size of the character data copied to dataPtr.
dataPtr is allocated on each call to GetCharDataPtr and must therefore be deallocated when it is not needed any more, or before GetCharDataPtr is called again.

dataPtr is deallocated by the call FreeMem(dataPtr, dataSize).

The structure of the data pointed to by dataPtr depends on the character type:

If fnt is attached to bitmapped characters, then dataPtr points to a bit block \((\text{symWidth} + 7) \div 8\) bytes per row.

If fnt is attached to stroke or outline characters, then dataPtr points to an array of points of the type MG.Point, if fnt contains "large" characters (the number of characters is dataSize \div SizeOf(MG.Point)) or the type MG.ShortPoint, if fnt contains "small" characters (the number of characters is dataSize \div SizeOf(MG.ShortPoint)).

If fnt is attached to Bézier characters, then dataPtr points to an array of Bézier elements of the type MG.Bezier (the number of characters is dataSize \div SizeOf(MG.Bezier)).

Data from GetCharDataPtr can be used as input to AddChar.

SV.Result
MGErr.invalidSymbols MG.deviceSymbols not allowed
MGErr.symbolOutOfRange ch is not in the font
MGErr.invalidSymbolsType MG.deviceSymbols are not allowed
MGErr.outOfMem not enough memory to allocate dataPtr

Errors are also reported from the virtual memory handler in VM.

TYPE
  PointArray = ARRAY[0..999] OF MG.Point;
VAR
  outlineFont, strokeFont : MG.Font;
  charWidth, charHeight, baseLine : INTEGER;
  polyDataPtr : ^PointArray;
  polySize : Word;
  ch : CHAR;
..
MG.SetFont(outlineFont);
FOR ch:= MG.CurFirstChar TO MG.CurLastChar DO BEGIN
    MG.GetCharDataPtr(outlineFont, ch,
    charWidth, charHeight, charBaseline,
    polySize, polyDataPtr);
    IF SV.Result = SV.ok THEN BEGIN
        MG.ClrWin; (* Draw the character polygon on screen *)
        MG.FillPolygon(polyData^, 0, polySize DIV SizeOf(MG.Point) - 1);
        (* Add character to the stroke font *)
        MG.AddChar(strokeFont,
        ch, charWidth, polySize, polyDataPtr^);
        FreeMem(polyDataPtr, polySize);
    END;
END; (*for*)

This is the same example as listed under GetCharData.

**DeleteChar**

```
P DeleteChar(fnt: Font;(*Io*)
    ch : CHAR);
```

Removes the character `ch` from the font attached to `fnt`. Character data is lost.

**SV.Result**
- **MGErr.invalidSymbols**: MG.deviceSymbols not allowed
- **MGErr.symbolOutOfRange**: `ch` is not found
- **MGErr.outOfMem**: out of memory for temporary buffer
  (needed when `fnt` is in virtual memory)

Errors are also reported from the virtual memory handler in VM.
**AddChar**

```pascal
P AddChar(  fnt    : Font; (*Io*)
             ch : CHAR;
             charWidth: INTEGER;
             dataSize : Word;
             VAR data              (*I*)
          );
```

Expands the font attached to `fnt` with data for the character `ch`. `charWidth` contain the width of the character and the character description data is given in `data` and encompasses `dataSize` bytes.

If there is already a character `ch` in the font, this will be replaced by the new character data. If the font has a fixed width the data in `charWidth` must correspond to this fixed value.

The structure of `data` depends on the font type:

- If `fnt` is attached to bitmapped characters, then `data` is a bit block `((symWidth + 7) DIV 8` bytes per row). 

- If `fnt` is attached to stroke or outline characters, then `data` is an array of points of the type `MG.Point`, if `fnt` contains "large" characters (`dataSize = <number of points> * SizeOf(MG.Point)`) or the type `MG.ShortPoint`, if `fnt` contains "small" characters (`dataSize = <number of points> * SizeOf(MG.ShortPoint)`).

- If `fnt` is attached to Bézier characters, then `data` is an array of Bézier elements of the type `MG.Bezier` (`dataSize = <number or elements> * SizeOf(MG.Bezier)`).

When calling `AddChar` the internal offset table is expanded to its maximum size, if this has not been done before. If the font is saved on disk, using `SaveFont`, the offset table is minimized. Therefore, when the font is reloaded from disk using `LoadFont`, the font will usually occupy less heap space than before.

Output data from `GetCharData` or `GetCharDataPtr` can be used as input data to `AddChar`. 

---

*Ref 15-22 MG - Fonts*

*MultiGraphics & Borland Pascal/Turbo Pascal*
SV.Result
MGErr.invalidSymbols    MG.deviceSymbols not allowed
MGErr.symbolOutOfRange  ch is not in the font
MGErr.invalidSymbolWidth invalid characters width (< 0,
127 if "small" characters,
or different from a fixed width)
MGErr.invalidSymbolData invalid dataSize (too large or not an
integer number of points or
Bézier elements)
MGErr.symbolTooLarge    dataSize too large
MGErr.outOfMem          out of memory for temporary buffer
(if fnt use virtual memory) or
for offset table

Errors are also reported from the virtual memory handler in VM.

CONST
    minInt = -32768;

TYPE
    PolyArray3 = ARRAY[0..2] OF MG.Point;
CONST
    paV: PolyArray3 = (
        (x: 0;y:30),
        (x:10;y:10),
        (x:20;y:30));

TYPE
    PolyArray5 = ARRAY[0..4] OF MG.Point;
CONST
    paX = PolyArray5(
        (x:0,y:10),
        (x:20,y:30),
        (x:minInt,y:minInt),
        (x:20,y:10),
        (x:0,y:30));

VAR
    fnt: MG.Font;
    ..
    MG.CreateFont(fnt,MG.strokeSymbols,
        FALSE,
        0,30,10);
    MG.AddChar(fnt,'V',20,SizeOf(paV),paV);
    MG.AddChar(fnt,'X',20,SizeOf(paX),paX);
    MG.SetFont(fnt);
    MG.SetPos(100,100); MG.DrawStr('VX');
    ..
    MG.SaveFont(fnt,'STROKE.COL');
    MG.DisposeFont(fnt);
The Current Font

**SetFont**

```pascal
P SetFont(fnt: Font);
```

Sets the current font in the current window. Prior to this `fnt` must have been attached to a font (normally using `LoadFont`, `CreateFont` or `CreateBitmapFont`).

When a window is created, the current font is set to the device's default font. Certain devices have no default font, in which case the current font is set to `nullFont`.

**SV.Result**

- `processWithoutWin`: the calling process has no window
- `invalidFont`: invalid font variable
- `invalidDeviceFont`: invalid device specific font

If a device specific font is activated, errors that are specific for the device can also be reported.

```pascal
VAR
  fnt : MG.Font;
  win1: MG.Window;
 ..
  MG.LoadFont(fnt,'SIMPLE.CSS',TRUE);
  MG.SetWin(win1); MG.SetFont(fnt);
```

**SetDefaultFont**

```pascal
P SetDefaultFont;
```

Sets the device's *default font* as the current font in the current window. The default font has no open font variable and has therefore its own "set" procedure in order to be activated.

When a window is created, the default font is assigned as the current font.
SV.Result
processWithoutWin the calling process has no window
invalidFont invalid font variable
invalidDeviceFont invalid device specific font

If a device specific font is activated, errors that are specific for the device can also be reported.

VAR
  fnt: MG.Font;
  ...
  MG.SetFont(fnt);
  MG.DrawStr('MENU');
  MG.SetDefaultFont;

CurFont

F CurFont: Font;

Returns a reference to the current font in the current window. If the font is missing, nullFont is returned.

This can be used to store the current font and then reset this if the font has been changed.

SV.Result
processWithoutWin the calling process has no window

If an error occurs, nullFont is returned.

VAR
  fnt, oldFnt: MG.Font;
  ...
  oldFnt := MG.CurFont;
  MGSetFont(fnt);
  ...
  MG.SetFont(oldFnt);

CurFontType

F CurFontType: SymbolsType;

Returns the font type for the current font in the current window.
SV.Result
processWithoutWin the calling process has no window
invalidSymbols the current font is missing or invalid

If an error occurs, **maxSymbolsType** is returned.

IF MG.CurFontType <> MG.bitmapSymbols THEN
  MG.TransformFont(fnt,2000,2500,0,0,0);
  ..

### CurCharWidth

**F CurCharWidth(ch: CHAR): INTEGER;**

Returns the width for the character `ch` in font of the current window.

SV.Result
processWithoutWin the calling process has no window
invalidFont the current font is missing or is invalid
deviceProcNotInUse the width routine is missing

If an error occurs, **MAXINT** is returned.

IF (MG.CurX + MG.CurCharWidth('A')) > MG.CurWinWidth THEN ..

### CurCharBaseline

**F CurCharBaseline: INTEGER;**

Returns the height for the base line in the font of the current window.

SV.Result
processWithoutWin the calling process has no window
invalidFont the current font is missing or is invalid
deviceProcNotInUse the base line height routine is missing

If an error occurs, **MAXINT** is returned.

IF MG.CurY - MG.CurCharBaseLine < 0 THEN
  MG.SetPos(0,MG.CurCharBaseLine);
  ..
CurCharHeight

F CurCharHeight: INTEGER;

Returns the height for the characters in the font of the current window.

SV.Result
- processWithoutWin: the calling process has no window
- invalidFont: the current font is missing or is invalid
- deviceProcNotInUse: the height routine is missing

If an error occurs, MAXINT is returned.

IF MG.CurX >= MG.CurWinWidth THEN
  MG.SetPos(0,
    MG.CurY - MG.CurCharHeight);

CurFirstChar

F CurFirstChar: CHAR;

Returns the first character in the current font and window.

SV.Result
- processWithoutWin: the calling process has no window
- invalidFont: the current font is missing or is invalid

If an error occurs, CHR(255) is returned.

FOR ch:= MG.CurFirstChar TO MG.CurLastChar DO ..

CurLastChar

F CurLastChar: CHAR;

Returns the last character in the current font and window.

SV.Result
- processWithoutWin: the calling process has no window
- invalidFont: the current font is missing or is invalid

If an error occurs, CHR(255) is returned.
GetCurFontInfo

\textbf{P} \textbf{GetCurFontInfo(VAR info: SymbolsInfo (*O*))};

Reads and returns in \textit{info} information about the current font in the current window. The information is contained in a record in accordance with the type \texttt{SymbolsInfo} (see Ref "MG - Symbols" under "Data Types").

\textbf{SV.Result}
\texttt{processWithoutWin} \hspace{1cm} the calling process has no window
\texttt{invalidSymbols} \hspace{1cm} the current font is missing or is invalid

If an error occurs, \textit{info} will contain "garbage"! See also \textbf{GetFontInfo}.

Strings

StrWidth

\textbf{F} \textbf{StrWidth(s: String): INTEGER;}

Returns the width in pixels of the string \textit{s} using the current font in the current window.

\textbf{SV.Result}
\texttt{processWithoutWin} \hspace{1cm} the calling process has no window
\texttt{invalidFont} \hspace{1cm} the current font is missing or is invalid
\texttt{deviceProcNotInUse} \hspace{1cm} device specific font has no width procedure

If an error occurs, \texttt{MAXINT} is returned.

\begin{verbatim}
MG.SetPos((MG.CurWinWidth - MG.StrWidth('CENTER')) DIV 2, 300);
MG.DrawStr('CENTER');
\end{verbatim}
Text Drawing

**DrawStr**

\[ \text{P DrawStr(s: String);} \]

Draws the string \( s \) starting in the current position and direction in the current window. The string is written using the current window mode, i.e. it is clipped against the current viewport, and drawn using the current foreground and background colors, color mask and drawing mode.

N.B. that **DrawStr** does not automatically perform new lines nor scrolling.

The current position is updated to a position after the last character in the string (no matter if this is visible or not).

**SV.Result**

- **processWithoutWin** the calling process has no window
- **invalidFont** the current font is missing or is invalid
- **deviceProcNotInUse** the device specific font has no width

Device specific errors are also reported as well as errors from virtual memory handling in **VM**. When an error occurs, the current position is not updated.

\[ \text{VAR} \]
\[ \begin{align*}
\text{bigFnt: MG.Font;} \\
\text{s : String} \\
.. \\
\text{s:= 'WARNING!';} \\
\text{MG.SetPos(10,20); MG.SetFont(bigFnt);} \\
\text{MG.MatchForeground(MG.red);} \\
\text{MG.MatchBackground(MG.cyan);} \\
\text{MG.SetColorMask(MG.CurMaxColorMask);} \\
\text{MG.SetDrawMode(MG.replace);} \\
\text{MG.\textbf{DrawStr}(s);} \\
\end{align*} \]
Borland Pascal/Turbo Pascal lacks support for concurrent processes and this part of MultiGraphics is therefore inapplicable here.
The **MGPat** unit contains a number of pre-declared *line styles* and *patterns* suitable for different types (raster, vector) of output devices with varying resolutions. **MGPat** defines *generic* standard line styles and standard patterns that are portable between output devices.

By using generic standard styles and standard patterns, the program's portability between different output devices is increased.

**Error Handling**

The unit **MGPat** does *not* use the *SuperVisor* technique for error handling. However, procedures in **MG** are called by "Set" routines in **MGPat**, and as a consequence error messages from **MG** should be checked.

**Program Examples**

The program **GAPat** draws a number of standard patterns.
Data Types

GenericLineStyle = (solid, dotted, 
                   dashed, dashedDotted);

The following generic line styles are defined:

solid       solid line (default)
dotted      dotted line
dashed      dashed line
dashedDotted alternating dots and dashes

See the function GenLineStyle and the procedure SetGenLineStyle.

GenericPattern = (solidPat, horiz, vert, 
                  diagUp, diagDown, 
                  cross, diagCross, 
                  halftone0, halftone1, halftone2, 
                  halftone3, halftone4, halftone5, 
                  halftone6, halftone7, halftone8, 
                  halftone9);

The following generic patterns are defined:

solidPat    solid (default)
horiz       horizontal lines
vert         vertical lines
diagUp       diagonal lines upwards to right
diagDown     diagonal lines downwards to left
cross        crossing horizontal and vertical lines
diagCross    crossing diagonal lines
halftone0    unfilled
halftone1..  increasing raster density (gray scale)
halftone8    solid (same as solidPat)
halftone9    solid (same as solidPat)

See the function GenPattern and the procedure SetGenPattern.
Density = 0..999;

Specifies the density of the "gray" scale. No raster corresponds to 0, while 999 corresponds to solid. The values in between these two extremes correspond to a "gray" scale of increasing density. How the density increases depends on the type of output device and the resolution of the current screen. See the procedure SetHalftone.

The remaining data types, that is to say

HardwareLineStyleStruc,LineStyle16Struc,
HardwarePatternStruc,
Pattern8Data, Pattern8Struc,
Pattern16Data, Pattern16Struc

are only of interest when defining your own patterns or for the construction of driver routines for output devices. See interface section and the section "User Defined Line Styles and Patterns" below and each driver documentation respectively.
Constants


\begin{verbatim}
unitName       'MGPat'
copyright      'DATABITEN'

The following identifiers are actually variables, but they are given specific values when initialized by MGPat and should not be modified.

\textbf{style16: ARRAY [GenericLineStyle] OF LineStyle;}

Line patterns of 16 bits in size for raster oriented output devices. They can be used when calling \texttt{SetLineStyle} in MG. For example:

\begin{verbatim}
MG.SetLineStyle(MGPat.style16[MGPat.dashed]);
\end{verbatim}

\textbf{dens8  : ARRAY [GenericPattern] OF Pattern;}
\textbf{sparse8: ARRAY [GenericPattern] OF Pattern;}
\textbf{dens16 : ARRAY [GenericPattern] OF Pattern;}

Defines 8x8 bit dense and sparse patterns, as well as a number of 16x16 patterns for raster oriented devices. The patterns can be used when calling \texttt{SetPattern} in MG. For example:

\begin{verbatim}
MG.SetPattern(MGPat.sparse8[MGPat.cross]);
\end{verbatim}

\textbf{bitPat8: ARRAY[0..15] OF Pattern;}

Defines 16 pre-declared 8x8 bit patterns for raster oriented devices. See fig 2. The patterns can be used when calling \texttt{SetPattern} in MG. For example:

\begin{verbatim}
MG.SetPattern(MGPat.bitPat8[10]);
\end{verbatim}
\end{verbatim}

\end{verbatim}
Fig 2: 8x8 patterns in `bitPat8`

Variables

See "Constants" above.
Procedures & Functions

Line Styles

GenLineStyle

F GenLineStyle(style: GenericLineStyle):LineStyle;

This function returns a line style of the type MG.LineStyle corresponding to generic style and suited to the current type of output device (raster, vector or hardware supported line drawing) and its resolution. The returned line style can be used as a parameter in calls to MG.SetLineStyle.

No errors can occur. SV.Result is not set.

USES MG,MGPat;
VAR
  style: MG.Pattern;
  ..
  style:= MGPat.GenLineStyle(MGPat.dashedDotted);
  MG.SetLineStyle(style);

SetGenLineStyle

P SetGenLineStyle(style: GenericLineStyle);

This procedure selects a line style suited to the type and resolution of the output device and makes it the current line style in the current window.

Replaces the call MG.SetLineStyle(MGPat.GenLineStyle(style));

SV.Result
MGErr.processWithoutWin the calling process has no window

MGPat.SetGenLineStyle(MGPat.dotted);

instead of

MG.SetLineStyle(MGPat.GenLineStyle(MGPat.dotted));
Patterns

**GenPattern**

\[ F \text{ GenPattern}(\text{pat: GenericPattern}): \text{Pattern}; \]

This function returns a pattern of the type `MG.Pattern` corresponding to the generic pattern `pat` and suited to the current type of output device (raster, vector, hardware supported pattern filling) and its resolution. The returned pattern can be used as a parameter in calls to `MG.SetPattern`.

No errors can occur. `SV.Result` is not set.

```pascal
VAR
  pat: MG.Pattern;
  ...
  pat := MGPat.GenPattern(MGPat.halftone5);
  MG.SetPattern(pat);
```

**SetGenPattern**

\[ P \text{ SetGenPattern}(\text{pat: GenericPattern}); \]

This procedure selects a pattern suited to the type and resolution of the output device and makes this the current pattern in the current window.

Replaces the call `MG.SetPattern(MGPat.GenPattern(pat))`;

`SV.Result`  
`MGErr.processWithoutWin` the calling process has no window

```pascal
MGPat.SetGenPattern(MGPat.diagUp);
```

instead of

```pascal
SetPattern(GenPattern(diagUp));
```
The SetHalftone function sets a halftone pattern (raster) with the desired density on a scale from 0 (empty) to 999 (solid). The halftone pattern then becomes the current pattern in the current window.

The interval between the increasing densities is dependent on the type and resolution of the current output device.

User Defined Line Styles and Patterns

To achieve the goal of maximum portability between different graphics devices, line styles and patterns in MultiGraphics are defined as hidden, abstract data types (LineStyle and Pattern). The concrete implementations of these types may differ between different devices. Devices with hardware support for a limited number of line styles and patterns often identify these styles and patterns using numbers, while a raster oriented device expects a bit pattern, etc. In spite of the different representations, we can always use the same "abstract" types when calling MultiGraphics procedures.

LineStyle and Pattern are always implemented as pointer types, i.e. they hold the addresses to the real data structures used for each device respectively. When you know the actual pattern structure used by a particular device, it is easy to create data of the type Pattern by doing the following operations:

```pascal
VAR
  patStruc: RECORD .... END;
  pat     : MG.Pattern;
 ..
  pat:= MG.Pattern(@patStruc);
MG.SetPattern(pat);
```

or directly
MG.SetPattern(MG.Pattern(@patStruc));

\section*{Line Style of Raster Devices}

Raster devices often use the data type \texttt{LineStyle16Struc} (see "Data Types" above) to implement line styles.

\begin{verbatim}
LineStyle16Struc = RECORD
   strucSize: Word;
   data     : Word;
END;
\end{verbatim}

The \texttt{strucSize} field gives the overall structure size, i.e. \texttt{SizeOf(LineStyleStruc)} (4 bytes). The line pattern is specified as an integer, where an included bit is drawn using the foreground color, while a missing bit is drawn using the background color. For example:

\begin{verbatim}
CONST
   brokenStruc : MGPat.LineStyle16Struc =
      (strucSize: 4;
       data     : $0FF3F);
VAR
   broken      : MG.Pattern;
   ...
   MG.SetLineStyle(MG.LineStyle(@brokenStruc));
   MG.DrawLine(100);
   broken:= MG.LineStyle(@brokenStruc);
   MG.SetLineStyle(broken);
   MG.DrawLine(100);
\end{verbatim}

The visible effect is the same in both cases.

\section*{Patterns for raster devices}

Raster devices with a resolution $\leq$ VGA 640x480 normally use the type \texttt{Pattern8Struc}, while devices with higher resolutions normally use \texttt{Pattern16Struc} (see "Data Types" above) to implement patterns.

\begin{verbatim}
Pattern8Data = ARRAY[0..7] OF Byte;
Pattern8Struc = RECORD
   strucSize: Word;
   rowSize  : Word;
   height   : Word;
   data     : Pattern8Data;
END;
\end{verbatim}
Pattern16Data = ARRAY[0..15] OF Word;
Pattern16Struc = RECORD
  strucSize: Word;
  rowSize  : Word;
  height   : Word;
  data     : Pattern16Data;
END;

The \texttt{strucSize} field gives the overall structure size, i.e. 
\texttt{SizeOf(Pattern\#Struc)} (4 bytes). \texttt{rowSize} gives the row size in bytes, 
while \texttt{height} gives the number of rows in the pattern. The pattern is speci-
fied as an array of bytes or integers, where an included bit is drawn using 
the foreground color, while a missing bit is drawn using the background 
color. For example

\texttt{CONST}
\begin{verbatim}
crossPatStruc:
  MGPat.Pattern8Struc =
    (strucsize: SizeOf(MGPat.Pattern8Struc);
     rowSize  : 1;
     height   : 8;
     data     :<$00,$18,$18,$7E,$7E,$18,$18,$00>);
\end{verbatim}
\texttt{VAR}
\begin{verbatim}
crossPat: MG.Pattern;
.. MG.SetPattern(MG.Pattern(@crossPatStruc));
MG.FillView;
crossPat:= MG.Pattern(@crossPatStruc);
MG.SetPattern(crossPat);
MG.FillView;
\end{verbatim}

The visible effect is the same in both cases.
The unit MG uses the RGB technique (red, blue, green values) in order to specify colors in a way that is device independent. The RGB model is well suited to the technique that is used to represent colors on monitor screens for example.

The RGB model is less well suited to our subjective perception of colors. Concepts such as hue, saturation and value are much more suitable yardsticks for describing our visual impressions. The initial letters of these three concepts are used to name the model used in MGCol, i.e. the HSV model.

Hue is specified with the help of a "color circle" and the unit of measurement used is the degree, within the range of 0..360. The primary color red corresponds to 0 degrees, green to 120 degrees and blue corresponds to 240 degrees.

Saturation specifies the density of the "color" within the range 0..999 (data type MG.Intensity). The lower the saturation, the more whiter/grayer the color appears. A saturation of 0 gives a gray or white color.

Value specifies the intensity of the color within the range 0..999 (data type MG.Intensity). The higher this value is the more intense the color appears to be. A value of 0 is completely black.

The unit MGCol has routines for converting between the HSV and RGB models.
MGCol also defines several constants with RGB colors. These may be used with the procedures MG.SetRGB and MG.SetRGBPalette as well as MG.MatchForeground, etc.

As a last feature, MGCol supports the setting of colors and palette according to the defaults used in Microsoft Windows 3.x. This is convenient, especially when you are using images, created by Windows based tools.

Error Handling

The unit MGCol uses the SuperVisor technique to handle errors. Errors are either handled automatically by error procedures (the default call is made to SV.GlbEP) or manually by checking SV.Result after calling procedures and functions in MGCol.

Constants

invalidHSV 1250

minHue 0
maxHue 360

The minimum and maximum values of hue (see data type ColorTone).

The following RGB colors are defined:

"Red"

darkRed, mediumRed, brightRed, lightRed, brick, cherry, pink, salmon

"Orange"

darkOrange, mediumOrange, brightOrange, lightOrange, orangeRed, yellowOrange, burntOrange, melon, tangerine
"Yellow"

darkYellow, mediumYellow, brightYellow, lightYellow, orangeYellow, greenYellow, amber, cream, gold, lemon, maize

"Green"

darkGreen, mediumGreen, brightGreen, lightGreen, yellowGreen, forest, grass, lime, olive, pine, seaGreen, springGreen

"Cyan"

darkCyan, mediumCyan, brightCyan, lightCyan, paleCyan, aquamarine, blueGreen, greenBlue, turquoise

"Blue"

darkBlue, mediumBlue, brightBlue, lightBlue, blueGray, cobaltBlue, cornFlower, skyBlue, violetBlue

"Purple"

darkPurple, mediumPurple, brightPurple, lightPurple, violet, lavender

"Magenta"

darkMagenta, mediumMagenta, brightMagenta, lightMagenta, paleMagenta, mulberry, orchid, redViolet
"Gray"
black, darkerGray, darkGray, mediumGray, lightGray, lighterGray, white, silver

"Materials"
darkBrown, brown, copper, mahogany, rust

"People"
darkSkin, mediumSkin, lightSkin, indianRed, apricot, peach

The Microsoft Windows Default Colors

mswBlack
mswRed
mswGreen
mswYellow
mswBlue
mswMagenta
mswCyan
mswDarkGray
mswLightGray
mswLightRed
mswLightGreen
mswLightYellow
mswLightBlue
mswLightMagenta
mswLightCyan
mswWhite

These 16 colors belong to the MS Windows default palette and are therefore appropriate when using the Windows palette as set by SetMSWindowsPalette. Note that the precise nuances usually differ from the MultiGraphics default colors MG.black..MG.white.
Data Types

ColorTone = minHue..maxHue;

This is the data type for hue. The values are given in degrees from 0 to 360 in accordance with the "color circle". The primary color red corresponds to 0 degrees, green to 120 degrees and blue to 240 degrees. Saturation and value are specified using the data type Intensity, which is defined in MG. Their values lie within the range of 0..999.

Procedures & Functions

Conversion Routines

<table>
<thead>
<tr>
<th>HSVtoRGB</th>
</tr>
</thead>
<tbody>
<tr>
<td>P HSVtoRGB( hue : ColorTone;</td>
</tr>
<tr>
<td>sat, value: Intensity;</td>
</tr>
<tr>
<td>VAR rgbCol : RGB (<em>O</em>));</td>
</tr>
</tbody>
</table>

Converts the color value in the HSV model to RGB values.

hue, is within the range 0..360.

sat, is within the range 0..999.

value, is within the range 0..999.

The result, given in rgbCol is an RGB value within the range 0..999 (see the unit MG for a definition of Intensity and RGB).

SV.Result

invalidHSV hue, sat or value outside the permitted range.

VAR
rgbCol: MG.RGB;

.. MGCol.HSVtoRGB(60,700,999,rgbCol);
MG.MatchForeground(rgbCol);
**RGBtoHSV**

```pascal
P RGBtoHSV( rgbCol : RGB;
VAR hue      : ColorTone;(*O*)
VAR sat,value: Intensity (*O*)));
```

Converts the color value in the RGB model to a HSV value.

The RGB value in `rgbCol` is given within the range 0..999 (see the unit `MG` for a definition of `Intensity` and `RGB`).

- `hue`, is within the range 0..360.
- `sat`, is within the range 0..999.
- `value`, is within the range 0..999.

**SV.Result**

`MGErr.invalidRGB` RGB values outside the permitted range.

```pascal
VAR
  rgbCol: MG.RGB;
  h     : MGCol.ColorTone;
  s,v   : MG.Intensity;
  ..
  rgbCol.r:= 999; rgbCol.g:= 500; rgbCol.b:= 0;
  MGCol.RGBtoHSV(rgbCol,h,s,v);
```

**Microsoft Windows Palette**

**SetMSWindowsPalette**

```pascal
P SetMSWindowsPalette;
```

This procedure sets the first 16 colors in the current palette according to the default palette used in Microsoft Windows 3.x. This is useful when you are using a lot of images created by Windows applications in your Multi-Graphics programs. To select colors in the Windows palette, you are recommended to use the predefined colors in the RGB constants `mswBlack..mswWhite`.
MultiGraphics has its own unique font format for bit mapped, stroke, outline and Bézier fonts. Such fonts can be loaded by calling `MG.LoadFont` or as symbols by calling `MG.LoadSymbols`. With the `MGFnt` unit, you can also use Borland's BGI stroke fonts (file type ."CHR").

You can load BGI font files as MultiGraphics fonts as well as MultiGraphics symbols. As soon as the fonts have been loaded, you can use them exactly as other fonts and symbols in MultiGraphics. For instance, you may save them as disk files (using `MG.SaveFont` or `MG.SaveSymbols`) and then reload them as ordinary fonts and symbols using `MG.LoadFont` or `MG.LoadSymbols`, respectively. These converted fonts and symbols will load faster than the original BGI fonts.

You may use the Borland Font Editor to create stroke fonts and then use them in MultiGraphics applications.

Program Example

GAFnt shows, among other things, how you can import a Borland BGI font.
Constants

invalidBGIFile 1350
Specific error code in MGFnt.

Procedures

Initializing

BGI Fonts

```
LoadBGI.Font (VAR fnt : MG.Font; (*O*)
    filename : String;
    adjustAspect: BOOLEAN;
    smallSize : BOOLEAN);
```

This procedure loads a stroke font in BGI format (normally of file type "CHR") and attach the font to the font variable fnt. The font filename and path are specified in filename. The adjustAspect parameter is used to specify if the font height should be scaled, to conform to the aspect ratio of the current device, or not. smallSize is used to specify if the font should be loaded as a "small" (TRUE) or "large" (FALSE) font (see "MG - Fonts" for more information on the different font formats).

You can use the loaded fonts exactly as other stroke fonts. If you use smallSize = FALSE, the scaling potential is increased compared to smallSize = TRUE. On the other hand, the "large" font will occupy more memory space.

SV.Result
invalidBGIFile invalid file format

Also reports other errors generated by MG, VM and plain I/O errors.
VAR
    fnt: MG.Font;

.. MGFnt.LoadBGIFont(fnt,'C:bp\bgi\trip.chr',TRUE,FALSE);
MG.SetFont(fnt);
MG.DrawStr('BGI font TRIP');
MG.DisposeFont(fnt);

**LoadBGISymbols**

```
P LoadBGISymbols(VAR symbs : MG.Symbols;(*O*)
    filename : String;
    adjustAspect: BOOLEAN;
    smallSize : BOOLEAN);
```

This procedure loads a stroke font in BGI format (normally of file type "CHR") and attach the font to the symbols variable `symbs`. The font file name and path are specified in `filename`. The `adjustAspect` parameter is used to specify if the symbols height should be scaled, to conform to the aspect ratio of the current device, or not. `smallSize` is used to specify if the symbols should be loaded as "small" (TRUE) or "large" (FALSE) symbols (see "MG - Symbols" for more information on the different symbol formats).

You can use the loaded symbols exactly as other stroke symbols. If you use `smallSize = FALSE`, the scaling potential is increased compared to `smallSize = TRUE`. On the other hand, the "large" symbols will occupy more memory space.

**SV.Result**

```
invalidBGIFile       invalid file format
```

Also reports other errors generated by MG, VM and plain I/O errors.
The unit **MGImg** is a comprehensive library of data types, procedures and functions to let you work with rectangular, raster images. Images are declared as variables of the type **Image**.

Images can be created "manually", be copied from screens (visible, virtual, etc.), be copied from other images or be saved to or loaded from image files on disk. Images can be reduced or enlarged ("zooming"). **MGImg** can save and load images in its own unique file format or in the portable PCX format.

The PCX format has its origins in *PC Paintbrush* (DOS and now Windows), the paint program from ZSoft and is supported by most PC graphics programs. Images can therefore be created with a paint program and then loaded into your **MultiGraphics** graphics applications. Or alternatively, you can create images using **MultiGraphics**, save these in the PCX format and then use them in other graphics programs.

The **Image** type in **MGImg** is implemented using **virtual screens** from the **MG** unit. Both monochrome and color virtual screens can be used. This implementation makes it possible to work with very large images, such as scanned images, and to modify images in the same way as normal screens. All window and drawing operations that work with virtual screens can be applied to images as well!

Images can also be given a **mask** to protect parts of the background when images are drawn. The masking technique can be seen as a generalization of the drawing mode **stamp** (see the unit **MG**), but in contrast to **stamp**, which always uses the background color in the mask, a masked image can
use all the colors, including the background color.

Using the masking technique, it is possible to define non-rectangular images, e.g. circular, lopsided or transparent images with holes. The technique is useful for creating *icons* for example.

Image masks are implemented using monochrome, virtual screens, which means that an image mask can be handled in the same way as the image itself.

If the device is palette oriented, images created will contain information about the image's palette in the RGB format. This means that it is possible to save an image with a particular palette, then load the image and automatically recreate the desired palette setting.

Images have a limited portability between various output devices. Images are portable if they:

```
```

* are monochrome or

* have the same pixel representation and do not have more color planes than the current output device.

### Error Handling

The unit MGImg uses the *SuperVisor* technique for error handling. Errors are either handled automatically by error procedures (the default call is to SV.GlbEP) or manually by checking SV.Result after calling procedures and function in MGImg.

### Images and Virtual Screens

The *Image* type is implemented using virtual screens in MG. Masking information is implemented using monochrome virtual screens. All operations on virtual screens can therefore be applied to images. Using the functions ImageScr and ImageMaskScr you have access to the image's virtual screen as well as the mask's screen. You can open a window inside an image, draw within the image, copy between images, printout the images etc..

The image does not need to comprise the whole of the virtual screen that is
linked to the **Image** variable. The part to be used is specified by the fields `imgLeftSkip`, `imgBottomSkip`, `width` and `height`.

**Masking Images**

By masking an image, you can protect the background on the screen where the image is to be drawn. The mask is only used when the current drawing mode is `replace`. Those areas of the monochrome mask screen that are set to one (device color 1) are protected. The image is only drawn where the mask is set to zero (device color 0).

An image with a mask is drawn in two stages:

1. First the mask is drawn using the **mask** mode. This results in a hole appearing in the graphics where the mask is set to zero. The other areas are not modified.

2. Secondly, the image is drawn using the **mix** mode. The image fills up the "holes" if the mask agrees with the image.

Figure 1 shows how an image *without* a mask is drawn in `replace` mode and figure 2 shows how an image *with* a mask is drawn in `replace` mode.

![Fig 1: Image without mask](image.png)
The Image Palette

A color image from a palette oriented device may contain information about the appropriate palette for the image. The palette is of the type `RGBPalette` (see unit `MG`). The palette is not set automatically when the image is drawn, because a change of palette affects everything in the output device, including previously drawn graphics. The palette can however be read using the function `ImageRGBPalette` and then be used to modify the current palette by calling `SetRGBPalette`.

Huge Images

Images created by `MGImg` are not limited to the size of the actual monitor screen, but can (if there is enough memory) be as large as the maximum size of the virtual screen. Images, just like virtual screens, use the virtual memory technique in the `VM` and `AutoVM` units.

By compiling to protected mode, or in real mode setting the current memory to extended XMS, expanded EMS or disk, very large scanned images can be handled with no problems.

Maximum image size in this implementation is 16384x16384 pixels!
Saving Images to Disk

Images can be saved in two different file formats. One format is unique for MultiGraphics and can store information about the image mask. The images are not compressed. The other type of format, the PCX format, is a standard format having good portability between different graphics programs. At present, the PCX format supports the following screen modes:

- CGA 640x200 2 colors,
- EGA 640x350 monochrome/16 colors,
- VGA 640x480 monochrome/16 colors,
- SuperVGA 800x600 monochrome/16 colors,
- MCGA 640x480 monochrome,
- MCGA 320x200 256 colors,
- SuperVGA (640x480, 800x600, 1024x768) 256 colors,
- 8514/A (640x480, 1024x768, etc) 256 colors,
- Hercules 720x350 monochrome.

Monochrome PCX files can be used on all raster devices. EGA/VGA PCX images can be used also on MCGA, SuperVGA and 8514/A 256 color modes.

The PCX format is compressed, but cannot contain mask information. If you want to mask PCX images, the mask must be saved in a separate PCX file. Using AddImageMask, the mask can then be linked to the particular image.

The file extension ".PCX" is used for whole images, while ".PCC" denotes clippings. PCX files contain palette information, which is not the case with PCC files.

Examples

GA7 reads and copies an image. GA8 loads and displays a PCX image. GAPCX displays every ".PCX" and ".PCC" it can find in the current directory. Try it in a 256 color mode! GAZoom shows zooming of a loaded PCX image. GAlmg flips and rotates images. In "Animation on the PC" you will find many examples on image animation.
Constants

unitName 'MGImg'
copyright 'DATABITEN'

invalidImage 1301
invalidImageScreen 1302
invalidMaskScreen 1303
invisibleImage 1304
noScreen 1305
invalidSkips 1306
invalidImagePart 1307
invalidImageFile 1308
invalidPCXfile 1309
readBlockError 1310

Error codes that are specific for the MGImg unit.

nullImage

Used to specify a non-existent image of Image type.

Data Types

Image

Data type for images.
The data type, ImageInfo, gives detailed information about images - see procedures CreateImage and GetImageInfo for information on how to use this.

width and height give the size of the image in pixels.

devFamily and devMode contain information about which output device the image is created for.

rgbPal is a pointer to the RGB palette the image is tailored for.

imgScr refers to the virtual screen the image is attached to.

imgLeftSkip and imgBottomSkip give the start position for the image's lower left corner in the virtual screen.

dynaImgScr indicates if the screen has been created dynamically (TRUE) or statically (FALSE). It is used by DisposeImage.

maskScr refers to any possible virtual mask screen that may be attached to the image. The mask screen marks out which areas of the target screen that are protected or not. Using the masking technique, it is possible to use silhouettes and other forms of non-rectangular images.

maskLeftSkip and maskBottomSkip indicate the start position of the image's lower left corner in the masking screen.

dynaMaskScr indicates if the screen has been created dynamically (TRUE) or statically (FALSE). It is used by DisposeImage.
Variables

revision: Word;
EP       : ErrProc;

Procedures & Functions

Create, Load and Save Images

CreateImage

P CreateImage(VAR img : Image; (*O*)
            VAR info: ImageInfo (*I*));

Creates a new image using the data in the variable info. The image is attached to the image variable img. The following fields in the info variable (see declaration of ImageInfo) must be given a value before the call is made:

imgScr, width, height, imgLeftSkip, imgBottomSkip and dynaImgScr.

imgScr must be attached to a virtual screen or a monochrome virtual screen.

maskScr can either be MG.nullScreen or be attached to a monochrome virtual screen. In the latter case, maskLeftSkip, maskBottomSkip and dynaMaskScr must also be defined. The mask information is used when drawing in replace mode.

rgbPal can be NIL or be attached to an RGB palette. CreateImage creates a copy of the original palette which means that, if so desired, it can be erased after the call has been made.

During the call, the data in info is checked to verify that it is consistent (e.g that info.imgLeftSkip < info.width).

The fields dynaImgScr and dynaMaskScr determine if the virtual screen for the image and the respective virtual screen for the mask are to be deallocated or not during a call to DisposeImage. If TRUE, the screens are
automatically deallocated for a **DisposeImage** call. If **FALSE**, the screens must be deallocated manually.

If the call fails, **img** is set to **nullImage**.

You should *not* call this procedure before calling a "Read" (ReadImage etc.) or "Load" (LoadImage, etc.) procedure. These procedures allocate the image memory automatically.

**SV.Result**

- **noScreen**
- **invalidMaskScreen**
  - invalid **info.imgScr**
  - **info.maskScr** non-monochrome virtual screen, invalid width, height, indent for mask screen
- **invalidImageScreen**
  - invalid width, height, indent for image

Also **MGErr.processWithoutWin**, **MGErr.outOfMem** and **MGErr.invalidWinSpec**.

```pascal
VAR
    img      : MGImg.Image;
    imgInfo  : MGImg.ImageInfo;
    ..
WITH imgInfo DO BEGIN
    width:= 200; height:= 100;
    devFamily:= MG.CurDeviceFamily;
    devMode:= MG.CurDeviceMode;
    MG.GetRGBPalette(rgbPal);
    MG.CreateVirtScr(imgScr,
                      200,100,
                      MG.CurMaxColorMask);
    imgLeftSkip:= 0; imgBottomSkip:= 0;
    maskScr:= MG.nullScreen;
END;(*with*)
MGImg.CreateImage(img,imgInfo);
MG.DisposeRGBPalette(imgInfo.rgbPal);
```

### ReadImage

**P ReadImage(VAR img : Image; (**O**)

- **width, height** : INTEGER)**

Creates an image attached to the variable **img** by reading the screen data from the current window, screen and output device. The output device must support virtual screens.
The image's lower left corner is defined by the current window position. The desired width and height are given by the parameters `width` and `height`. The position and size are however clipped against the window's current viewport (no error message).

The current window mode (position, etc.) is not modified by the call.

The image's color information (e.g. the number of color planes when several color planes are used) is limited to the current color mask. If, for example, the screen has 4 color planes (i.e. max. color mask = 15) and the current color mask = 5 (1 + 4), the image will have 2 color planes, obtained from the screen's first and third color planes.

The image variable `img` contains information about the palette of the current output device (RGB format) when the call is made. The palette can be read using the function, `ImageRGBPalette`. Also see `ReadImageNoPal`.

If the call fails, `img` is set to `nullImage`.

**SV.Result**

`invisibleImage` the image is completely outside the current viewport.

Also `MGErr.processWithoutWin`, `MGErr.outOfMem`, `MGErr.invalidScreenType`, `MGErr.invalidWinSize`, `MGErr.invalidMaxColorMask`.

In the event of insufficient memory, we recommend compiling to protected mode, or in real mode, using expanded EMS memory, extended XMS memory or disk. See the units `VM` and `AutoVM`.

```pascal
VAR
  img: MGImg.Image;
  ...
  MG.SetColorMask(5);  read planes 0 & 2 (1 + 4 = 5) only
  MG.SetPos(20,30);
  MGImg.ReadImage(img,200,100);
  MG.SetPos(300,25);
  MG.SetColorMask(MG.CurMaxColorMask); all planes
  MGImg.DrawImage(img);
```
ReadImageNoPal

\[ P \text{ ReadImageNoPal(VAR img : Image; \texttt{(*O*)}}\]
\[ \text{width, height: INTEGER}); \]

The same as ReadImage but without reading the image palette. This speeds up reading of small images on devices with huge palettes as 256 colors MCGA, SuperVGA and 8514/A.

ReadMonoImage

\[ P \text{ ReadMonoImage(VAR img : Image; \texttt{(*O*)}}\]
\[ \text{width, height: INTEGER}); \]

Creates a monochrome image attached to the variable img by reading the screen data from the current window, screen and device. The current device must support monochrome virtual screens.

The image's lower left corner is defined by the current window position. The desired width and height are given by the parameters width and height. Position and size are however clipped against the window's current viewport (no error message).

The current window mode (position, etc.) is not modified by the call.

The screen's color information is interpreted so that the current background color is translated to the device color 0, while all the other colors are given device color 1. Only color planes contained within the current color mask are read.

If the call fails, img is set to nullImage.

SV.Result invisibleImage the image is completely outside the current viewport.

Also MGErr.processWithoutWin, MGErr.outOfMem, MGErr.invalidScreenType, MGErr.invalidWinSize.

In the event of insufficient memory, we recommend compiling to protected mode, or in real mode, using expanded EMS memory, extended XMS memory or disk. See the units VM and AutoVM.
VAR
  img: MGImg.Image;
 ..
  MG.SetPos(20,30); MG.MatchBackground(MG.blue);
  MGImg.ReadMonoImage(img,100,50);
  MG.SetPos(250,100);
  MG.MatchForeground(MG.yellow);
  MG.MatchBackground(MG.red);
  MGImg.DrawImage(img);

**LoadImage**

P LoadImage(VAR img : Image; (*O*)
             fname: String);

Loads an image from the image file given in `fname`. The image file must have the unique file format created by `SaveImage`. The expression `fname` can contain the path + filename. The loaded image is attached to the image variable `img`.

Image files created by `SaveImage` contain image data and information about the type of device. They can also store an image mask and an image palette. The palette is not set automatically, instead you must make the call `MG.SetRGBPalette(MGImg.ImageRGBPalette(img))`.

The file is opened in *read-only* mode.

If loading fails, `img` is set to `nullImage`.

```
SV.Result
invalidImageFile   invalid image file format
invalidImageScreen image information is missing
< 0                 IO error reported from the file handling system
```

Also `MGErr.processWithoutWin`, `MGErr.invalidColorMask`, `MGErr.outOfMem`, `MGErr.invalidScreenType`, `MGErr.invalidWinSize` and errors from the units `VM` and `AutoVM`.

VAR
  img: MGImg.Image;
 ..
  SV.SetEP(FALSE);
  MGImg.LoadImage(img,'C:\img\flower.img');
  IF SV.Result <> SV.ok THEN ..
LoadPCXImage

P LoadPCXImage(VAR img : Image; (*O*)
    fname: String);

Loads an image from the image file given in \texttt{fname}. The image file must use the PCX or PCC format created by \texttt{SavePCXImage} or by drawing programs such as PC Paintbrush and Windows Paintbrush. The \texttt{fname} expression can contain the path + filename. The loaded image is attached to the image variable \texttt{img}.

Image files saved using \texttt{SavePCXImage}, or other programs that create PCX or PCC files, contain not only image data but also information about the type of the graphics mode and any palette information as well. The image palette is \textit{not} set automatically on loading the image file however. In contrast to the unique image format in \texttt{MultiGraphics}, PCX and PCC files do \textit{not} contain information about masking images.

PCX files, created by PC Paintbrush, contain palette information while PCC files do not. PCC files are used as "clip outs", while PCX files contain complete color information. \texttt{LoadPCXImage} can read both formats.

\texttt{MGImg} supports PCX/PCC files for monochrome monitors, CGA 640x200, EGA, VGA, SuperVGA, 8514/A and MCGA with 256 colors. 256 colors graphics modes can read 16 colors PCX/PCC files as well. All raster devices can use monochrome PCX/PCC files.

The file is opened in \textit{read-only} mode.

If loading fails, \texttt{img} is set to \texttt{nullImage}.

\texttt{SV.Result}
\texttt{invalidPCXfile} invalid PCX format
\texttt{< 0} IO error reported from the file handling system

Also \texttt{MGErr.processWithoutWin}, \texttt{MGErr.invalidColorMask}, \texttt{MGErr.outOfMem}, \texttt{MGErr.invalidScreenType}, \texttt{MGErr.invalidWinSize} errors from the unit \texttt{VM}.
VAR
  img: MGImg.Image;

..  
VM.SetVM(EmsVM.vmm); (*use EMS to store image*)
SV.SetEP(FALSE);
MGImg.LoadPCXImage(img,'C:\pbrush\mandril.pcx');
IF SV.Result <> SV.ok THEN ..
MG.SetRGBPalette(MGImg.ImageRGBPalette(img));

### CloneImage

P CloneImage(  sourceImg: Image;
                VAR cloneImg : Image (*O*) );

Creates a copy of the image in sourceImg and attach the image to the variable cloneImg. The original image is not changed. The "clone" is stored in the current virtual memory.

You can use CloneImage to make several modified copies of an image, while the original image is kept intact, or to move an image from conventional memory to extended XMS, expanded EMS or disk in the event of insufficient memory, etc.

If copying fails cloneImg is set to nullImage.

SV.Result
invalidImage  invalid original image

Also MGErr.processWithoutWin, MGErr.invalidColorMask,
MGErr.outOfMem, MGErr.invalidScreenType,
MGErr.invalidWinSize errors from the unit VM.

### AddImageMask

P AddImageMask(img : Image; (*Io*)
                imgMask : Screen;
                leftSkip,bottomSkip: INTEGER);

Attaches a screen mask to an existing image in img.

The mask is specified as a monochrome screen, imgMask, with the lower left corner indented in accordance with leftSkip, bottomSkip. The width of the mask is assumed to be the same as that of the image. If the image img is drawn using replace mode, areas where the mask device color = 0 will erase
the background while areas with mask device color = 1 will save the background. The image information is then added using the drawing mode **mix**.

With **DisposeImage**, the screen mask is not deallocated automatically. It is important therefore that this is deallocated using **DisposeScr(imgMask)**.

**SV.Result**
- **invalidImage**: invalid original image in **img**
- **invalidSkips**: invalid indent
- **invalidMaskScreen**: invalid monochrome screen mask, e.g. invalid monochrome screen, or size

### SaveImage

```pascal
P SaveImage(img : Image;
            fname: String);
```

Saves the existing image attached to the variable **img** in an image file on disk. Path and file name are to be specified in **fname**. The image is stored in **MultiGraphics**’s unique image format.

The saved image can store not only image data, but also information about which graphics mode the image is intended for, the image mask (if any) and the image palette.

**SV.Result**
- **invalidImage**: invalid image variable
- **< 0**: IO error reported from the file handling system

Errors from the unit **VM** are also reported.

### SavePCXImage

```pascal
P SavePCXImage(img : Image;
                fname: String);
```

Saves the existing image attached to the variable **img** in an image file on disk. Path and file name are to be specified in **fname**. The image is stored in standard PCX/PCC format. We recommend using '.PCX' or '.PCC' as file name extensions.

The saved image can store image data, information about the graphics mode
Ref 20-16  MGIimg - Images

and image palette. PCX files cannot however contain information about masking.

MGIimg supports PCX/PCC files for monochrome monitors, CGA 640x200, EGA/VGA 16 color, SuperVGA, 8514/A and MCGA with 256 colors.

SV.Result
invalidImage  invalid image variable
readBlockError  error in copying from the image's screen
< 0  IO error reported from the file handling system

Errors from the units VM and AutoVM are also reported.

**DisposeImage**

P DisposeImage(VAR img: Image (*IO*));

Deletes the image information and deallocates the dynamic memory that has been attached to the variable img. If the call was successful the image variable is set to nullImage.

The virtual screen with image information linked to img is deallocated only if it is labeled as dynamic (see the field ImageInfo.dynaImgScr). This is always true if the image has been created by calling ReadImage, ReadMonoImage, CloneImage, ZoomImage, ZoomCopyImage, LoadImage or LoadPCXImage. If the image has been created by calling CreateImage, the virtual screen is automatically deallocated if the dynaImgScr field is set to TRUE. If the field is FALSE, the virtual screen must be deallocated "manually" by calling DisposeScr in MG.

The monochrome virtual screen with mask information that may be attached to img, is deallocated only if it is labeled as dynamic (see the field ImageInfo.dynaMaskScr). This is true if the mask information has been created by calling LoadImage, CloneImage, ZoomImage or ZoomCopyImage. If the mask has been attached to the image using a call to CreateImage, the mask screen is automatically deallocated if the field dynaMaskScr is set to TRUE. If the field is FALSE or the mask has been attached to the image by calling AddImageMask, the mask must be deallocated "manually" by calling DisposeScr in MG.

Images use dynamic (virtual) memory. In order to save memory, it is important to
delete images that are no longer used, especially when image variables have been
declared locally in procedures. If a local image variable is not deleted before the
procedure is completed, dynamic memory is lost.

It is important to call \texttt{DisposeImage} before the device that owns the virtual screen
attached to the image is closed. Otherwise you will get an error when calling \texttt{Dispose-
Image}.

\begin{verbatim}
SV.Result
invalidImage invalid image variable

MGErr.processWithoutWin and MGErr.invalidScreen errors from the
unit VM are also reported.

VAR
  img: MGImg.Image;
  ..
  MGImg.LoadPCXImage(img, 'house.pcx');
  ..
  MGImg\texttt{.DisposeImage(img);}
\end{verbatim}

Modifying Images

\begin{verbatim}
\textbf{FlipImageHor} \texttt{DL(R)}

P FlipImageHor(img: Image (*Io*));

Flips (mirrors) an image attached to \texttt{img} in the horizontal direction - the
"mirror" is held vertically.

SV.Result
invalidImage invalid image

Also MGErr.processWithoutWin, MGErr.outOfMem,
MGErr.invalidScreenType and errors from the unit VM.
\end{verbatim}

\begin{verbatim}
\textbf{FlipImageVert} \texttt{DL(R)}

P FlipImageVert(img: Image (*Io*));

Flips (mirrors) an image attached to \texttt{img} in the vertical direction - the
"mirror" is held horizontally.
\end{verbatim}
SV.Result
invalidImage  invalid image

Also MGErr.processWithoutWin, MGErr.outOfMem, MGErr.invalidScreenType and errors from the unit VM.

RotImage180 DL(R)

P RotImage180(img: Image (*Io*));

Rotates an image attached to img by an angle of 180 degrees - the image is turned upside-down.

SV.Result
invalidImage  invalid image

Also MGErr.processWithoutWin, MGErr.outOfMem, MGErr.invalidScreenType and errors from the unit VM.

RotImage90 DL(R)

P RotImage90(sourceImg: Image; (*Io*)
clockwise: BOOLEAN);

Rotates an image attached to img by an angle of 90 degrees - in clockwise direction if clockwise is set to TRUE, in anticlockwise direction if clockwise is FALSE.

The command will swap the image width and height!

SV.Result
invalidImage  invalid image
Also MGErr.processWithoutWin, MGErr.outOfMem, MGErr.invalidScreenType and errors from the unit VM.

### RotCopyImage90

```pascal
P RotCopyImage90(    sourceImg: Image;
    clockwise: BOOLEAN;
    VAR rotImg   : Image (*O*));
```

Creates a new and rotated copy, `rotImg`, of the original image attached to `sourceImg`. The copy is rotated by an angle of 90 degrees - in clockwise direction if `clockwise` is set to TRUE and in anticlockwise direction if `clockwise` is FALSE.

The original image in `sourceImg` is not modified.

SV.Result
invalidImage invalid source image

Also MGErr.processWithoutWin, MGErr.outOfMem, MGErr.invalidScreenType and errors from the unit VM.

### ZoomImage

```pascal
P ZoomImage(img       : Image;
    zoomWidth,
    zoomHeight: INTEGER;
    mixMode   : DrawMode);`
```

Enlarges or reduces the image attached to `img`. The width of the zoomed image is specified in `zoomWidth` and its height in `zoomHeight`. When reducing the width or height compared to the source image, the merged pixels are combined according to the drawing mode in `mixMode`.

The most convenient drawing modes are replace, which overwrites information when reducing and is fast, but may lose image information, mix and mask, which protect the image information better but is slower. See ZoomScr in "MG - Screens" for more details on zooming and drawing modes.

*In contrast to ZoomCopyImage, the original image in sourceImg is modified when calling ZoomImage!*
SV.Result
invalidImage  invalid image

Also MGErr.processWithoutWin, MGErr.invalidColorMask,
MGErr.outOfMem, MGErr.invalidScreenType,
MGErr.invalidWinSize and errors from the unit VM.

VAR
  img: MGImg.Image;
  ..
  MGImg.LoadPCXImage(img, 'EARTH.PCX');
  MGImg.ZoomImage(img,
      MG.CurWinWidth,
      MG.CurWinHeight,
      MG.replace);
  MG.SetPos(0, 0); MGImg.DrawImage(img);
  MGImg.DisposeImage(img);

**ZoomCopyImage**

P ZoomCopyImage(   sourceImg : Image;
                  zoomWidth,
                  zoomHeight: INTEGER;
                  mixMode   : DrawMode;
               VAR zoomImg : Image (*O*));

Creates an enlarged or reduced copy of the original image attached to
sourceImg, and attaches the copy to the image variable zoomImg. The
width of the copy is specified in zoomWidth and its height in zoomHeight.
When reducing the width or height compared to the source image, the
merged pixels are combined according to the drawing mode in mixMode.

The original image in sourceImg is not modified! The zoomed image in
zoomImg is created on calling ZoomCopyImage and need not be created
in advance.

As the source image is not modified, this image may be scaled many times.
ZoomCopyImage is well suited for adapting images to different screen
resolutions.

The most convenient drawing modes are replace, which overwrites informa-
tion when reducing and is fast, but may lose image information, mix and
mask, which protect the image information better but is slower. See
ZoomScr in "MG - Screens" for more details on zooming and drawing
modes.
SV.Result
invalidImage invalid original image

Also MGErr.processWithoutWin, MGErr.invalidColorMask,
MGErr.outOfMem, MGErr.invalidScreenType,
MGErr.invalidWinSize errors from the unit VM.

VAR
  img, zoomImg: MGImg.Image;

  MG.SetPos(100,100);
  MGImg.ReadImage(img,200,150);
  MGImg.ZoomCopyImage(img,100,125,MG.mix,zoomImg);
  MG.SetPos(0,0); MGImg.DrawImage(zoomImg);
  MGImg.DisposeImage(zoomImg);
  MGImg.ZoomCopyImage(img,640,480,MG.replace,zoomImg);
  MGImg.SavePCXImage(zoomImg,'ZOOM.PCX');
  MGImg.DisposeImage(zoomImg);
  MGImg.DisposeImage(img);

Drawing Images

<table>
<thead>
<tr>
<th>DrawImage</th>
<th>DL(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P DrawImage(img: Image);</td>
<td></td>
</tr>
</tbody>
</table>

Draws an image attached to the image variable img in the current window, screen and output device. The image's lower left corner is placed in the current window position.

The image is drawn with regard to the window's current drawing mode, color mask and, for monochrome images, the device colors for the current foreground and background colors. The image is clipped against the current viewport.

If the current drawing mode is replace, image mask information, if present, is used to protect parts of the background graphics in the current window. The background is erased where the mask device color = 0 and the background is protected where the mask device color = 1 and the image device color = 0.

When the call is made, the image variable must be active and the image's virtual screen type must be compatible with the screen type of the current...
device. The images and the current screen are compatible provided the current screen is of the raster type and that:

- the image is monochrome or
- the image and the current screen have the same pixel representation (bits per pixel) and the current screen's (max) color mask can hold the maximum color mask for the image.

MCGA, SuperVGA or 8514/A images (8 bits per pixel) are therefore not compatible with a VGA screen (1 bit per pixel and color plane). VGA and EGA screens, on the other hand, are compatible (i.e. they have the same pixel representation).

The latter condition is fulfilled, for example, if the color mask for the current screen is 15 (e.g. EGA and VGA) and the image has a maximum color mask <= 15 (i.e. 1, 2, 3 or 4 color planes).

16 colors PCX/PCC images when loaded are converted to 256 colors format if the current device is a 256 colors device.

If the image is monochrome, those areas of the image having a device color = 0 are drawn using the output window's current background color. Those areas of the image having a device color = 1, are drawn using the current window's foreground color.

If the image is not monochrome, the image's device color is retained when drawing. However, those device colors that are not included in the current color mask are masked off. Moreover, the result is affected by the current drawing mode.

SV.Result
invalidImage
invalid image variable

Also MGErr.processWithoutWin, MGErr.incompatibleDevices, MGErr.deviceProcNotInUse, errors from the unit VM.

VAR
colorImg, monoImg: MGImg.Image;
win1,win2 : MG.Window;
.
MG.SetWin(win1);
MG.SetBackground(1); MG.SetPos(0,0);
MGImg.ReadMonoImage(monoImg,100,100);
MG.SetColorMask(4 + 8); MG.SetPos(100,0);
MGImg.ReadImage(colorImg,100,100);
MG.SetColorMask(15);
MG.SetWin(win2); MG.SetPos(0,0);
MG.SetBackground(4); MG.SetForeground(10);
MG.DrawImage(monoImg);
MG.SetPos(100,0); MG.SetColorMask(15);
MG.SetDrawMode(MG.stamp);
MG.DrawImage(colorImg);

MG.DisposeImage(monoImg);
MG.DisposeImage(colorImg);

<table>
<thead>
<tr>
<th>DrawImagePart</th>
<th>DL(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P DrawImagePart(img</td>
<td></td>
</tr>
<tr>
<td>leftSkip, bottomSkip:</td>
<td></td>
</tr>
<tr>
<td>width, height:</td>
<td></td>
</tr>
</tbody>
</table>

Draws the indicated part of the image attached to the image variable `img` in the current window, screen and device. The lower left corner of that part of the image is placed in the current window position.

`leftSkip` and `bottomSkip` give the lower left corner of that part of the image to be drawn. The coordinates are given in the image's own coordinates (for the screen window in the screen variable linked to `img`). (0,0) corresponds to the lower left corner. The area's width and height is given by `width` and `height`.

Otherwise, the function of `DrawImagePart` is the same as `DrawImage`. 
DrawImagePart is useful when you want to draw a part of a larger image (e.g. a scanned image) or want to store a library of image symbols in a larger image.

**SV.Result**

- invalidImage: invalid image variable
- invalidImagePart: invalid parameters leftSkip, bottomSkip, width or height

Also **MGErr.processWithoutWin**, **MGErr.incompatibleDevices**, **MGErr.deviceProcNotInUse**, errors from the unit VM.

```pascal
VAR
  iconlib: MGImg.Image;
  icon   : INTEGER;

  MGImg.LoadPCXImage(iconlib,'C:\pbrush\ICONS.PCX');
  MG.SetPos(100,50); icon:= 10;
  MGImg.DrawImagePart(iconlib,
                      50*(icon MOD 7),
                      50*(icon DIV 7),
                      50,50);
```

**Fig 3: Draw part of an image**
Image Data

GetImageInfo

\[
P \text{GetImageInfo}( \ img \ : \ Image; \\
\quad \text{VAR} \ info: \ \text{ImageInfo} \ (*O*));
\]

Gives complete information about the image attached to the variable \text{img}. After the call, the \text{info} record variable contains information about the size of the image, the type of graphics device, the image palette, as well as information about the virtual screen that holds the image data and, if present, the image mask screen. See data type \text{ImageInfo} for more details.

SV.Result

invalidImage  \quad \text{invalid image variable}

VAR

\begin{verbatim}
  img : MGImg.Image;
  info: MGImg.ImageInfo;
 ..
  (* Delete image and check if automatic deallocation of screens *)
  MGImg.GetImageInfo(img,info);
  IF NOT info.dynaImgScr THEN MG.DisposeScr(info.imgScr);
  IF NOT info.dynaMaskScr THEN MG.DisposeScr(info.maskScr);
  MGImg.DisposeImage(img);
\end{verbatim}

ImageOK

\[
F \ \text{ImageOK}(\ img: \ Image): \ BOOLEAN;
\]

Returns \text{TRUE} if the image variable \text{img} is attached to a valid image, otherwise \text{FALSE}. 
Ref 20-26  

### IsMonoImage

**F** IsMonoImage(img: Image): BOOLEAN;

Returns **TRUE** if the image variable `img` is attached to a monochrome image, otherwise **FALSE**.

### ImageScr

**F** ImageScr(img: Image): Screen;

Returns the virtual screen that is attached to the image variable `img`.

The function can be useful when you want to modify an image in `img`. All operations for screen and window handling as well as drawing operations function just as for ordinary virtual screens and monochrome virtual screens.

### SV.Result

**invalidImage**  
invalid image variable

Returns **FALSE** if the specified image is invalid.

```pascal
VAR
    img1, img2: MGimg.Image;
    w, h      : INTEGER;

    MGImg.LoadPCXImage(img1,'beauty.pcx');
    MGImg.LoadPCXImage(img2,'logo.pcc');
    (* Make img1 the current screen *)
    MG.SetScr(MGImg.ImageScr(img1));
    w:= MG.CurWinWidth;
    h:= MG.CurWinHeight;
    (* Put a frame around the image *)
    MG.MatchForeground(MG.brown);
    MG.SetPos(0,0); MG.DrawBox(w,h);
    (* Create a window in img1 *)
    MG.CreateWin(win,1,1,w - 2,h - 2);
    MG.SetWin(win);
    (* Draw img2 on top of img1 *)
    MG.SetPos(img2 on top of img1 *)
    MG.SetPos(10,10);
```
MG.SetDrawMode(MG.stamp);
MG.SetBackground(0); MG.MatchForeground(MG.yellow);
MGImg.DrawImage(img2);
(* Save modified img1 *)
MGImg.SavePCXImage(img1,'beauty1.pcx');

**ImageMaskScr**

```pascal
F ImageMaskScr(img: Image): Screen;
```

Returns the monochrome virtual mask screen that perhaps is attached to the image variable in `img`.

This function can be useful when you want to work with the mask screen in `img`. You can use all operations for screen and window handling as well as drawing operations.

**SV.Result**

invalidImage invalid image variable.

If the image variable is invalid, `MG.nullScreen` is returned.

**ImageWidth**

```pascal
F ImageWidth(img: Image): INTEGER;
```

Returns the width in pixels of the image attached to the image variable `img`.

**SV.Result**

invalidImage invalid image variable

If the image variable is invalid, `MAXINT` is returned.

**ImageHeight**

```pascal
F ImageHeight(img: Image): INTEGER;
```

Returns the height in pixels of the image attached to the image variable `img`.

**SV.Result**

invalidImage invalid image variable
If the image variable is invalid, \texttt{MAXINT} is returned.

\begin{description}
\item[\texttt{ImageMaxColorMask}] \end{description}

\begin{verbatim}
F ImageMaxColorMask(img: Image): DeviceColor;
\end{verbatim}

Returns the maximum color mask for the virtual screen attached to the image variable \texttt{img}.

\begin{description}
\item[SV.Result]
\item[\texttt{invalidImage}] invalid image variable
\end{description}

If the image variable is invalid, \texttt{maxDeviceColor} is returned.

\begin{description}
\item[\texttt{ImageRGBPalette}] \end{description}

\begin{verbatim}
F ImageRGBPalette(img: Image): RGBPalette;
\end{verbatim}

Returns the image's RGB palette, if the image is created on a palette oriented device. The information can be used in order to adapt the current palette to the image palette.

If the image does not contain a palette (perhaps created on a true color device or without palette), the value \texttt{NIL} is returned.

\begin{description}
\item[SV.Result]
\item[\texttt{invalidImage}] invalid image variable
\end{description}

If the image variable is invalid, \texttt{NIL} is returned.

\begin{verbatim}
VAR
  img: MGImg.Image;
  ..
  MG.SetRGBPalette(MGImg.ImageRGBPalette(img));
  MGImg.DrawImage(img);
\end{verbatim}
MGIW - Integer World Coordinates

The unit MGIW is used to enhance the capabilities of windows in MG by using integer based world coordinates.

Window coordinates in MG are pixel oriented and thus device dependent. The lower left corner in a window always has the coordinates (0,0). The x coordinates increase to the right while y coordinates increase upwards.

World coordinates in MGIW, on the other hand are integer based but device independent. The same world coordinates can be used regardless of the number of pixels of a specific device, screen or window. World coordinates are individually attached to windows and each window can have its own world coordinates. Both the start value and the direction of world coordinates can be chosen freely in the integer range -32768 to +32767.

For example, the x axis can vary between 0..10000 while the y axis varies between -5000..+5000.

When a window with integer world coordinates is created, the window's world coordinates are set to the range of 0..10000 for both the x and y coordinates.

Integer world coordinates make it easy to write graphics programs that are independent of the size and resolution of the device. Using world coordinates, data can automatically be scaled when drawing.
MGIW versus MGW

The unit MGW also supports world coordinates, but these are floating-point based, instead of integer based. Compared to MGW, MGIW has the following advantages:

+ faster transformations, especially compared to MGW in the absence of a math co-processor (80x87 or 486DX),
+ integer based coordinates save memory, especially when using polylines, polygons and polybéziers.

At the same time, MGIW has the following drawbacks compared to MGW:

- less freedom when selecting the world coordinates (MGIW only accepts the integer interval -32768..32767).
Identical Names in MG and MGIW

MGIW uses the same data types for points and Béziers as MG, i.e. MG.Point and MG.Bezier. Angles are given in decidegrees, exactly the same way as in MG.

In MGIW you will find the equivalent to all those procedures in MG that have parameters for position, distance, or viewport, or which affect the current position. The procedures in MGIW have the same name and parameter lists as the procedures in MG.

We have therefore shortened the documentation of procedures in MGIW that have equivalents in MG. For more information about functions and error codes, we refer you to the documentation for the "sister" procedures in MG.

In contrast to MGW, MGIW lacks procedures to specify directions. There is, for example, a MGW.SetDir but no MGIW.SetDir. Here we use the "standard" MG.SetDir instead.

In order to avoid any misunderstandings using Pascal, it is important to place the units in the right order in the USES list. If you want to use MGIW procedures, MGIW ought to be placed after MG in the USES list. For example:

USES MG,MGIW,MGImg;
..  
SetPos(0.2,0.9); (* refers to MGIW.SetPos *)
MG.SetPos(100,200);

However, by always qualifying procedures with their unit names, confusions can be avoided.

Integer World Coordinates Capability

Before a window can use procedures in MGIW, its capabilities must be enhanced using the integer world coordinates capability. This can be done in one of the following three ways:

1. Individually for each window (screen window coordinates):

   MG.CreateWin(win1,50,50,200,100);
   MG.EnhanceWin(win1,MGIW.cap);
2. Individually for each window (the screen window has the integer world coordinates capability):

```pascal
MG.EnhanceWin(MG.CurWin, MGIW.cap);
MGIW.CreateWin(win1, 500, 500, 2000, 1000);
MGIW.CreateWin(win2, 3000, 500, 2000, 1000);
```

3. Collectively for all new windows in a screen:

```pascal
MG.EnhanceScr(scr, MGIW.cap);
MG.CreateWin(win1, 500, 500, 2000, 1000);
MG.CreateWin(win2, 300, 50, 2000, 1000);
```

When the new capability has been installed, the window's world coordinates are set to the range of 0..10000 for both the x and y coordinates. These world coordinates are then said to be *normalized*.

The current position is set to (0,0) and the current viewport is assigned to the whole window.

By calling `SetWorld`, new world coordinates can be attached to the window.

### Window Mode and World mode

A window with the integer world coordinates capability has, like all windows, a current *window mode* which is defined in `MG`, as well as a window based *world mode* that is defined in `MGIW`.

The procedures in `MGIW` support the current world mode, but also automatically update the current window mode in `MG`.

Data for the current position and viewport is thus to be found on two levels - on the one hand expressed in integer world coordinates and degrees in `MGIW`, and on the other hand expressed in device dependent integer coordinates in `MG`.

For example, if you set a new position using `MGIW.SetPos` and then call `MG.GetPos`, it will be found that the device dependent integer position has also been updated.

A word of warning is necessary here; when calling procedures in `MG` that
update the current position or viewport, the equivalent position or viewport in MGIW will not be updated. The two coordinate systems are "out of step" with each other (i.e. they are not synchronized)!

If a window has been enhanced with integer world coordinates, it is important that all calls that affect the current position or viewport are made to procedures in MGIW!

For example, MG.DrawStr updates the current position in MG but not in MGIW. MGIW.DrawStr updates the current position in both MG and MGIW.

As the integer coordinates must be in the interval -32768..32767 it is likely that you occasionally will submit integer world coordinates that after transformation would fall outside the integer range! This results in the error MGIW.invalidPos when calling for instance MGIW.SetPos.

If you want to know the absolute integer world limits of the current integer world coordinates, calling MGIW.GetWorldLimits will give you all information about that. By relying on this information, you will also avoid transformation errors (using 8514/A, it is also necessary to use software clipping, as the hardware clipping does not work outside a limited interval - see appendix "MG8514 - 8514/A").

Concepts and Definitions

By a window coordinate is meant a coordinate specified in device dependent window coordinates. By a window position is meant a position (x,y) specified in window coordinates.

By a integer world coordinate is meant a coordinate specified in the current integer world coordinate system. By a integer world position is meant a position (x,y) specified in integer world coordinates.

By a window unit is meant the unit of length (measurement) for window coordinates, normally specified in pixels.

By a integer world unit is meant the unit of length for integer world coordinates. A world coordinate system can have different world units in the x and y directions.
Constants

unitName 'MGIW'
copyright 'DATABITEN'

Unit name and copyright.

invalidWorld 1123
invalidPos 1124
invalidRect 1126

These are specific error codes for the MGIW unit.

Data Types

IntWorldMode = RECORD
  worldx1, worldx2, worldy1, worldy2: INTEGER;
  viewX0, viewY0,
  viewWidth, viewHeight : INTEGER;
  xpos, ypos              : INTEGER;
END;

Data types for integer world mode for windows that have world coordinates capabilities.

Procedure variables

EP: SV.ErrProc;
Local error procedure for the MGIW unit.

revision: CARDINAL;
Current revision for the implementation of the unit MGIW.

cap: Capability;
Capability variable for integer world coordinates. This is used as a para-
meter to MG.EnhanceWin and MG.EnhanceScr to give windows and screens world coordinates capabilities.

Procedures & Functions

Creating World Windows

CreateWin

\[
P \text{CreateWin(VAR win : Window; \quad (*O*)} \\
\quad \text{scrX0,scrY0 : INTEGER;}
\quad \text{width,height : INTEGER});
\]

This procedure creates a window and enhances it with the integer world coordinates capability. \((\text{scrX0,scrY0})\) specifies the position for the window's lower left corner, expressed in the screen window's world coordinates. \(\text{width}\) and \(\text{height}\) gives the window's width and the height respectively, expressed in the screen window's integer world units.

The current screen window must be enhanced prior to this with the integer world coordinates capability.

\text{MGIW.CreateWin} combines the effect of \text{MG.CreateWin} and \text{MG.EnhanceWin} with the ability to express the window's position in integer world coordinates and its size in integer world units.

A window with the integer world coordinates capability is removed, just like other windows, by a direct call to \text{MG.DisposeWin}, or indirectly by calling \text{MG.DisposeScr} or even \text{MG.CloseDevice}.

See \text{MG.CreateWin} and \text{MG.EnhanceWin} for more details.

\text{SV.Result}
\text{MGErr.capDontExist} \quad \text{the current window has no integer world coordinates capability}

Errors are also reported in accordance with \text{MG.CreateWin} and \text{MG.EnhanceWin}.
Ref 21-8  MGIW - Integer World Coordinates

MG.SetScrWin;
MG.EnhanceWin(MG.CurWin,MGIW.cap);
MGIW.SetWorld(0,1000,0,1000);
MGIW.CreateWin(win1,50,50,200,100);
MGIW.CreateWin(win2,250,50,200,100);

Window Handles

GetWinHandlesAtPos

P GetWinHandlesAtPos( scrX, scrY: INTEGER;
VAR handles : WindowHandleSet (*O*));

This procedure search for and returns in the variable parameter handles, the set of all window handles (0..254) in the current screen whose corresponding windows enclose the given position (scrX, scrY). The position is given in screen integer world coordinates. If this position is not located inside any window of the screen, the empty set is returned.

SV.Result
MGErr.processWithoutWin  the calling process has no window
MGErr.capDontExist  current window has no integer world coordinates capability

When an error occurs, the empty set is returned.

The Current World Mode

World Mode

GetWorldMode

P GetWorldMode(VAR mode: IntWorldMode (*O*));

Returns the current window's integer world mode in the variable mode.

The data type IntWorldMode is defined above under "Data Types".

This procedure is normally used with SetWorldMode to save the current integer world mode and then restore it.
SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no integer world coordinates capability

VAR
  win : MG.Window;
  oldWorldMode: MGIW.IntWorldMode;
  winMode : MG.WindowMode;

  MG.EnhanceWin(win,MGIW.cap);
  MGIW.GetWorldMode(oldWorldMode);
  MG.GetWinMode(winMode);

  MG.SetWinMode(oldWinMode);
  MGIW.SetWorldMode(oldWorldMode);

SetWorldMode

P SetWorldMode(VAR mode: IntWorldMode (*I*));

This procedure sets the current window's integer world mode. It also updates the current window mode in MG with regard to position and viewport.

SetWorldMode does not verify that the information in the variable mode is correct. The programmer must check this himself.

The procedure is normally used with GetWorldMode in order to store and restore a particular world mode.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no integer world coordinates capability

World Coordinates

SetWorld

P SetWorld(x1,x2,y1,y2: INTEGER);

This procedure defines the current integer world coordinate system for the
current window.

x1 specifies the world coordinate for the window's left hand side, while x2 specifies the world coordinate for the window's right hand side. x1 <> x2. Note that x2 < x1 is allowed.

y1 specifies the world coordinate for the window's lower side, while y2 specifies the world coordinate for the window's top side. y1 <> y2. Note that y2 < y1 is allowed.

All subsequent calls by procedures in MGIW will use the new the integer world coordinates.

The current window must have the integer world coordinates capability.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no integer world coordinates capability
invalidWorld invalid world coordinates

MG.SetWin(win);
MGIW.SetWorld(-5000,10000,5000,-5000);

GetWorld

P GetWorld(VAR x1,x2,y1,y2: INTEGER (*O*));

This procedure returns the current window's integer world coordinate system.

x1 specifies the world coordinate for the window's left hand side, while x2 specifies the world coordinate for the window's right hand side.

y1 specifies the world coordinate for the window's lower side, while y2 specifies the world coordinate for the window's top side.

The current window must have the integer world coordinates capability.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no integer world coordinates capability
GetWorldLimits

\[ \text{P GetWorldLimits(VAR xmin, xmax, ymin, ymax: INTEGER (*O*));} \]

This procedure returns the absolute limits for the current window's integer world coordinates.

\( \text{xmin} \) specifies the smallest acceptable world coordinate for x values, while \( \text{xmax} \) specifies the largest acceptable world coordinate for x values.

\( \text{ymin} \) specifies the smallest acceptable world coordinate for y values, while \( \text{ymax} \) specifies the largest acceptable world coordinate for y values.

If a drawing operation specifies a world coordinate outside these limits, the result will be an error during transformation to device coordinates.

The current window must have the integer world coordinates capability.

Converting Coordinates

DevX

\[ \text{F DevX(x: INTEGER): INTEGER;} \]

This function converts the integer world coordinate \( x \) in the current window to a device specific window coordinate.

The current window must have the integer world coordinates capability.

If an error occurs, \text{MAXINT} is returned.
\[ x := \text{MGIW.DevX}(2125); \]

\textbf{DevY}

\begin{verbatim}
F DevY(y: INTEGER): INTEGER;

This function converts the integer world coordinate \( y \) in the current window to a device specific window coordinate.

The current window must have the integer world coordinates capability.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no integer world coordinates capability

If an error occurs, \texttt{MAXINT} is returned.
\end{verbatim}

\textbf{DevWidth}

\begin{verbatim}
F DevWidth(w: INTEGER): INTEGER;

This function converts the width \( w \) in integer world units to a device specific width in window units.

The current window must have the integer world coordinates capability.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no integer world coordinates capability

If an error occurs, \texttt{MAXINT} is returned.
\end{verbatim}

\texttt{VAR}
\begin{verbatim}
  w: INTEGER;
  ..
  w := \text{MGIW.DevWidth}(1000);
\end{verbatim}
**DevHeight**

F DevHeight (h: INTEGER): INTEGER;

This function converts the height h in *integer world units* to a device specific height in window units.

The current window must have the integer world coordinates capability.

SV.Result
MGEerr.processWithoutWin       the calling process has no window
MGEerr.capDontExist            current window has no integer world coordinates capability

If an error occurs, MAXINT is returned.

**WorldX**

F WorldX(x: INTEGER): INTEGER;

This function converts the window coordinate x to an integer world coordinate.

The current window must have the integer world coordinates capability.

SV.Result
MGEerr.processWithoutWin       the calling process has no window
MGEerr.capDontExist            current window has no integer world coordinates capability

If an error occurs, MAXINT is returned.

MGIW.SetPos(MGIW.WorldX(100),MGIW.WorldY(100));
**WorldY**

\[ F \text{ WorldY}(y: \text{ INTEGER}): \text{ INTEGER}; \]

This function converts the window coordinate \( y \) to an integer world coordinate.

The current window must have the integer world coordinates capability.

**SV. Result**

- MGErr.processWithoutWin: the calling process has no window
- MGErr.capDontExist: current window has no integer world coordinates capability

If an error occurs, \texttt{MAXINT} is returned.

**WorldWidth**

\[ F \text{ WorldWidth}(w: \text{ INTEGER}): \text{ INTEGER}; \]

This function converts the width \( w \) in window units to a width in integer world units.

The current window must have the integer world coordinates capability.

**SV. Result**

- MGErr.processWithoutWin: the calling process has no window
- MGErr.capDontExist: current window has no integer world coordinates capability

If an error occurs, \texttt{MAXINT} is returned.

\[
\text{MGIW.SetPos(1000 + MGIW.WorldWidth(100), 5000);} 
\]

**WorldHeight**

\[ F \text{ WorldHeight}(h: \text{ INTEGER}): \text{ INTEGER}; \]

This function converts the height \( h \) in window units to a height in integer world units.

The current window must have the integer world coordinates capability.
SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no integer
world coordinates capability

If an error occurs, MAXINT is returned.

Position

<table>
<thead>
<tr>
<th>SetPos</th>
</tr>
</thead>
<tbody>
<tr>
<td>P SetPos(x, y: INTEGER);</td>
</tr>
</tbody>
</table>

This procedure sets the new current position in the current window. x and y are specified in integer world coordinates. See also MG.SetPos.

SV.Result
MGErr.capDontExist current window has no integer
world coordinates capability
invalidPos invalid position

MGIW.SetPos(5000, 6000);

<table>
<thead>
<tr>
<th>SetRelPos</th>
</tr>
</thead>
<tbody>
<tr>
<td>P SetRelPos(dx, dy: INTEGER);</td>
</tr>
</tbody>
</table>

This procedure sets the new position relative to the current position in the current window. dx and dy are specified in integer world units. See also MG.SetRelPos.

SV.Result
MGErr.capDontExist current window has no integer
world coordinates capability
invalidPos invalid position

MGIW.SetRelPos(1000, -1000);
MovePos

\[ \text{P \ MovePos(dist: INTEGER);} \]

This procedure moves the current position the distance \( \text{dist} \) in the current direction. \( \text{dist} \) is specified in \textit{x integer world units}. See also \textbf{MG.MovePos}.

\textbf{SV.Result}

\textbf{MGErr.capDontExist} \hspace{1cm} \text{current window has no integer world coordinates capability}

\textbf{invalidPos} \hspace{1cm} \text{invalid position}

\texttt{MG.SetDir(450);}
\texttt{MGIW.MovePos(3000);}

GetPos

\[ \text{P \ GetPos(VAR x,y: INTEGER (*O*));} \]

This procedure reads the current position in the current window. \( x \) and \( y \) are expressed in integer world coordinates. See also \textbf{MG.GetPos}.

\textbf{SV.Result}

\textbf{MGErr.capDontExist} \hspace{1cm} \text{current window has no integer world coordinates capability}

CurX

\[ \text{F \ CurX: INTEGER;} \]

This function returns the current \( x \) position in the current window. The position is expressed in \( x \) integer world coordinates. See also \textbf{MG.CurX}.

\textbf{SV.Result}

\textbf{MGErr.capDontExist} \hspace{1cm} \text{current window has no integer world coordinates capability}

If an error occurs, \texttt{MAXINT} is returned.
VAR
  x: INTEGER;
  ..
  x := MGIW.CurX;

CurY

F CurY: INTEGER;
This function returns the current y position in the current window. The
position is expressed in y integer world coordinates. See also MG.CurY.

SV.Result
MGErr.capDontExist    current window has no integer
                      world coordinates capability

If an error occurs, MAXINT is returned.

GetFastXY

P GetFastXY(VAR x, y: INTEGER (*IO*));

Based on the given x and y values, this procedure returns proposals for the
nearest x and y values which would result in a higher drawing speed. x and
y are specified in integer world coordinates. See also MG.GetFastXY.

SV.Result
MGErr.capDontExist    current window has no integer
                      world coordinates capability

VAR
  x, y: INTEGER;
  ..
  x := 3000; y := 8000;
  MGIW.GetFastXY(x, y);
  MGIW.SetPos(x, y);
Viewport

**SetView**

```pascal
P SetView(x0,y0, width, height: INTEGER);
```

This procedure sets the current window's viewport. \(x0\) and \(y0\) are specified in integer world coordinates, while \(width\) and \(height\) are given in integer world units. See also **MG.SetView**.

When a window capability has been installed, the viewport is set to the whole window.

**SV.Result**

- **MGErr.capDontExist** current window has no integer world coordinates capability
- **MGErr.invalidViewSpec** invalid boundaries for the viewport

**MGIW.SetView**(2000,0,5000,1000);

---

**GetView**

```pascal
P GetView(VAR x0,y0, width, height: INTEGER (*O*));
```

This procedure reads the location and size of the current window's viewport. \(x0\) and \(y0\) are expressed in integer world coordinates while \(width\) and \(height\) are given in integer world units. See also **MG.GetView**.

**SV.Result**

- **MGErr.capDontExist** current window has no integer world coordinates capability
Draw Graphics

Points

**DrawPoint**

```pascal
procedure DrawPoint (x, y: INTEGER;
                     color: DeviceColor);
```

This procedure draws a point at integer world position \((x,y)\) with device color `color`, using the current color mask and drawing mode. If the position lies outside the current viewport, nothing is drawn. The current position and colors are not modified.

See also `MG.DrawPoint`.

**SV.Result**
- `MGErr.capDontExist` current window has no integer world coordinates capability
- `invalidPos` invalid position
- `MGErr.invalidDeviceColor` invalid device color

**PointDeviceColor**

```pascal
function PointDeviceColor (x, y: INTEGER): DeviceColor;
```

This function returns the *device color* for a point (pixel) at integer world position \((x,y)\) in the current window. The specified position must be within the current viewport.

See also `MG.PointDeviceColor`.

**SV.Result**
- `MGErr.capDontExist` current window has no integer world coordinates capability
- `invalidPos` invalid position
Straight Lines

**DrawLineTo**

```pascal
P DrawLineTo(x, y: INTEGER);
```

This procedure draws a straight line from the current position to \((x, y)\) in the current window. \(x\) and \(y\) are specified in integer world coordinates. \((x, y)\) becomes the new current world position. See also `MG.DrawLineTo`.

**SV.Result**
- `MGErr.capDontExist`: current window has no integer world coordinates capability
- `invalidPos`: invalid final position

```pascal
MGIW.SetPos(-200, 1500);
MGIW.DrawLineTo(-50, 2000);
```

**DrawLineRel**

```pascal
P DrawLineRel(dx, dy: INTEGER);
```

This procedure draws a straight line relatively from current position \((x_0, y_0)\) to \((x_0 + dx, y_0 + dy)\). The displacement is given in integer world units. \((x_0 + dx, y_0 + dy)\) becomes the new current position. See also `MG.DrawLineRel`.

**SV.Result**
- `MGErr.capDontExist`: current window has no integer world coordinates capability
- `invalidPos`: invalid final position

```pascal
MGIW.DrawLineRel(1000, -1000);
```

**DrawLine**

```pascal
P DrawLine(dist: INTEGER);
```

This procedure draws a straight line using the length \(dist\) from the current position and in the current drawing direction. The distance is given in \(x\) integer world units. See also `MG.DrawLine`.

**SV.Result**
- `MGErr.capDontExist`: current window has no integer world coordinates capability
- `invalidPos`: invalid final position

```pascal
MGIW.DrawLine(1000, -1000);
```
SV.Result
MGErr.capDontExist  current window has no integer
world coordinates capability
invalidPos  invalid final position

MG.SetDir(1350);
MGIW.DrawLine(5000);

Polylines

DrawPolyline

P DrawPolyline(VAR polyline; (*I*)
  first,last: Word);

This procedure draws a polyline based on the current position and the array
of relative points, of the type MG.Point, that are given in the variable
parameter polyline. The points are specified in integer world units. The
current world position is set relative to the last point used in the array
(current position + polyline[last]).

A special point, MG.nullPoint, is used to separate different polylines in the
array of points from each other.

See also MG.DrawLine.

SV.Result
MGErr.capDontExist  current window has no integer
world coordinates capability
MGErr.tooManyPoints  too many points

Rectangles

DrawBox

P DrawBox(width, height: INTEGER);

This procedure draws a rectangle (box) with sides width and height, with
the starting corner in the current window position and the base in the
x direction. The width and height are specified in x integer world units and
y integer world units, respectively. The current world position remains
unchanged. See also MG.DrawBox.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability

MGIW.DrawBox(3000,1000);

**DrawRect**

P DrawRect(width, height: INTEGER);

This procedure draws a rectangle with sides width and height, with the starting corner in the current window position and the base in the current drawing direction. The lengths of the sides are specified in x integer world units. The current world position remains unchanged. See also MG.DrawBox.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability

MG.SetDir(450);
MGIW.DrawRect(3000,1000);

Circles, Arcs and Circle Sectors

**DrawCircle**

P DrawCircle(radius: INTEGER);

This procedure draws a circle with its center in the current integer world position using the radius in the parameter radius. The radius is specified in x integer world units. See also MG.DrawCircle.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability
MGErr.invalidRadiusOrAxis invalid radius

MGIW.DrawCircle(3000);
DrawArc

\texttt{P \textit{DrawArc}(radius, centralAngle: \textbf{INTEGER});}

This procedure draws an \textit{arc} with its center in the current integer world position, the starting angle in the current drawing direction, the radius \textit{radius} and the central angle \textit{centralAngle}. The radius is specified in \textit{x integer world units} and the central angle in \textit{decidegrees}. The current world position is not changed. The current drawing direction is assigned to the arc's final angle. See also \texttt{MG.DrawArc}.

\texttt{MGIW.DrawArc(3000,450);}

DrawSector

\texttt{P \textit{DrawSector}(radius, centralAngle: \textbf{INTEGER});}

This procedure draws a \textit{circle sector} with the center in the current integer world position, the starting angle in the current drawing direction, the radius \textit{radius} and the central angle \textit{centralAngle}. The radius is specified in \textit{x integer world units} and the central angle in \textit{decidegrees}. The current position is not changed. The current direction is assigned to the sector's final angle. See also \texttt{MG.DrawSector}.

\texttt{MGIW.DrawSector(3000,450);}
Ellipses, Elliptical Arcs and Sectors of Ellipses

**DrawEllipse**

```pascal
P DrawEllipse(a,b,majorAxisAngle: INTEGER);
```

This procedure draws an *ellipse* with its center in the current integer world position, the direction of the major axis in accordance with `majorAxisAngle`, the length of the major axis `a` and the length of the minor axis `b`. The major axis and minor axis are specified in *integer world units*. The direction of the major axis is specified in *decimal degrees*. The current world position and direction are not changed. See also `MG.DrawEllipse`.

**SV.Result**

- `MGErr.capDontExist` current window has no integer world coordinates capability
- `MGErr.invalidRadiusOrAxis` invalid axes
- `invalidAngle` invalid angle

```pascal
MGIW.DrawEllipse(3000,2000,900);
```

**DrawEllipseArc**

```pascal
P DrawEllipseArc(a,b,majorAxisAngle: INTEGER;
                centralAngle    : INTEGER);
```

This procedure draws an *elliptical arc* with its center in the current integer world position, the starting angle in the current drawing direction, the major axis `a`, the minor axis `b`, the direction of the major axis `majorAxisAngle` and the central angle `centralAngle`. The major and minor axes are specified in *integer world units*. The directions of the major axis and the central angle are specified in *decimal degrees*. The current world position is not changed. The current direction is assigned to the arc's final angle. See also `MG.DrawEllipseArc`.

**SV.Result**

- `MGErr.capDontExist` current window has no integer world coordinates capability
- `MGErr.invalidRadiusOrAxis` invalid axes
- `invalidAngle` invalid angle

```pascal
MGIW.DrawEllipseArc(3500,2200,300,-200);
```
**DrawEllipseSector**

\[
P \text{DrawEllipseSector}(a,b,\text{majorAxisAngle}: \text{INTEGER}; \\
\text{centralAngle} : \text{INTEGER});
\]

This procedure draws a *sector of an ellipse* with its center in the current integer world position, the starting angle in the current drawing direction, the major axis \(a\), the minor axis \(b\), the direction of the major axis \text{majorAxisAngle} and the central angle \text{centralAngle}. The major and minor axes are specified in *x integer world units*. The directions of the major axis and the central angle are specified in *decidegrees*. The current world position is not changed. The current direction is assigned to the sector's final angle. See also \text{MG.DrawEllipseSector}.

**SV.Result**

\text{MGErr.capDontExist} \quad \text{current window has no integer world coordinates capability}

\text{MGErr.invalidRadiusOrAxis} \quad \text{invalid axes}

\text{invalidAngle} \quad \text{invalid angle}

\text{MGIW.DrawEllipseSector}(3500, 2200, 300, -200);

**Bézier Curves**

**DrawPolyBezzer**

\[
P \text{DrawPolyBezzer}((\text{VAR polyBez; (* ARRAY OF MG.Bezier*)} (*I*) \\
\text{first,last: Word});)
\]

Draws a *polybézier* curve relative the current position and the array of *relative* Bézier elements, of the type \text{MG.Bezier}, found in the variable parameter \text{polyBez}. The polybézier starts from the Bézier element with the index \text{first} and ends with the element with the index \text{last}, where \text{last} >= \text{first}.

The control points of the Bézier curve is given in integer world units. The new current position is the point (relative to the current start position) which is given as the last control point of the Bézier element with index \text{last}.

**SV.Result**

\text{MGErr.capDontExist} \quad \text{current window has no integer world coordinates capability}
Patterns

**FillRect**

P FillRect(x0,y0,width,height: INTEGER);

This procedure fills the specified rectangular area in the current window using the current pattern. \((x0,y0)\) specifies the rectangle's lower left corner, expressed in integer world coordinates. **width** and **height** specify the rectangle's width and the height in integer world units. See also MG.FillRect.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability
MGErr.invalidRect invalid rectangle

MGPat.SetGenPattern(MGPat.diagUp);
MGIW.FillRect(-4000,2000,8000,4000);

**FillPolygon**

P FillPolygon(VAR vertices; (* ARRAY OF MG.Point*) (*I*)
                     first,last: Word);

This procedure fills a polygon, based on the current position and the array of relative point data, of the type MG.Point, that is given in the variable parameter **vertices**. The polygon starts at the point with the index **first** and ends at the point with the index **last**, where **last** \(>= \) **first**. Points are specified in integer world units.

A special point, MG.nullPoint, is used to fill separate polygons and to prevent the inner parts of a polygon from being filled.

See also MG.FillPolygon.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability
FillCircle

P FillCircle(radius: INTEGER);

This procedure fills a circle with the current pattern. The circle's center is at the current position and the radius is \texttt{radius}. The radius is specified in \textit{x integer world units}. The current position is not changed. See also MG.FillCircle.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability
MGErr.invalidRadiusOrAxis invalid radius

MGPat.SetGenPattern(MGPat.diagUp);
MGIW.FillCircle(3000);

FillSector

P FillSector(radius, centralAngle: INTEGER);

This procedure fills a circle sector with the current pattern. The center is at the current window position, the starting angle in the current drawing direction, the radius is \texttt{radius} and the central angle \texttt{centralAngle}. The radius is specified in \textit{x integer world units}. The central angle is specified in \textit{decidegrees}. The current position and direction are not changed. See also MG.FillSector.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability
MGErr.invalidRadiusOrAxis invalid radius
MGErr.invalidAngle invalid angle

MG.SetDir(450);
MGIW.FillSector(3000,450);
FillEllipse

\textbf{P FillEllipse}(a,b,\text{majorAxisAngle}: \text{INTEGER});

This procedure fills an \textit{ellipse} with the current pattern. The ellipse's center is at the current world position. \textbf{a} specifies the ellipse's major axis and \textbf{b} the minor axis. \textbf{majorAxisAngle} gives the direction of the major axis. The major and minor axes are specified in \textit{x integer world units}. The direction of the major axis is specified in \textit{decidegrees}. The current position and direction are not changed. See also \texttt{MG.FillEllipse}.

\textbf{SV.Result}

\texttt{MGErr.capDontExist} \hspace{1cm} \text{current window has no integer world coordinates capability}

\texttt{MGErr.invalidRadiusOrAxis} \hspace{1cm} \text{invalid axes}

\texttt{invalidAngle} \hspace{1cm} \text{invalid angle}

FillEllipseSector

\textbf{P FillEllipseSector}(a,b,\text{majorAxisAngle}: \text{INTEGER};
\text{centralAngle} : \text{INTEGER});

This procedure fills a \textit{sector of an ellipse} with the current pattern. The ellipse's center lies in the current window position, the sector's starting angle in the current drawing direction, the major axis is given in \textbf{a}, the minor axis in \textbf{b}, the direction of the major axis in \textbf{majorAxisAngle} and the sector's central angle in \textbf{centralAngle}. The major and minor axes are specified in \textit{x integer world units}. The directions of the major axis and the central angle are specified in \textit{decidegrees}. The current position and direction are not changed. See also \texttt{MG.FillEllipseSector}.

\textbf{SV.Result}

\texttt{MGErr.capDontExist} \hspace{1cm} \text{current window has no integer world coordinates capability}

\texttt{MGErr.invalidRadiusOrAxis} \hspace{1cm} \text{invalid axes}

\texttt{invalidAngle} \hspace{1cm} \text{invalid angle}
Filling of Polybézier Curves

**FillPolyBezier**

```
P FillPolyBezier(VAR polyBez; /* ARRAY OF MG.Bezier */) (*I*)
  first, last: Word);
```

Fills a polybézier curve relative the current position and the array of *relative* Bézier elements, of the type **MG.Bezier**, found in the variable parameter `polyBez`. The polybézier starts from the Bézier element with the index `first` and ends with the element with the index `last`, where `last >= first`.

*The border line is also drawn (in contrast to MGIW.FillPolygon), if the line style is not set to MG.nullLineStyle.*

The current position and drawing direction are not changed.

The only permitted start index of the polybézier array is 0. The coordinates of the control points do not specify absolute coordinates, instead these are given *relative to the current integer world position*.

**SV.Result**
- **MGErr.capDontExist** current window has no integer world coordinates capability
- **outOfMem** can't create internal temporary conversion buffer
- **tooManyPoints** internal buffer is filled up

**Copying**

**CopyRect**

```
P CopyRect (source : Window;
  x0, y0 : INTEGER;
  width, height : INTEGER);
```

This procedure copies the graphics in a rectangular area, located in the window `source`, to the current window. The original rectangle's lower left corner is specified using `(x0, y0)` in `source` integer world coordinates. The width and height are specified using `width` and `height` in `source` integer world units. The copy's lower left corner is placed at the current position.
and window. See also MG.CopyRect.

**SV.Result**

MGErr.capDontExist  
current window has no integer world coordinates capability

MG.SetWin(win2);  
MGIW.SetPos(0,0);  
MGIW.CopyRect(win1,0,0,5000,10000);

**Symbols**

**CurSymbolWidth**

\[ P \text{ CurSymbolWidth}(\text{symbol}: \text{Word}): \text{INTEGER}; \]

This procedure returns the width for the symbol with the number `symbol` in the current window's symbol library. The width is given in *x* integer world units. See also MG.CurSymbolWidth.

**SV.Result**

MGErr.capDontExist  
current window has no integer world coordinates capability

If an error occurs, MAXINT is returned.

**CurSymbolHeight**

\[ P \text{ CurSymbolHeight}(\text{symbol}: \text{Word}): \text{INTEGER}; \]

This procedure returns the height for the symbol with the number `symbol` in current window's symbol library. The height is given in *y* integer world units. See also MG.CurSymbolHeight.

**SV.Result**

MGErr.capDontExist  
current window has no integer world coordinates capability

If an error occurs, MAXINT is returned.
GetHotSpot

P GetHotSpot (symbol: Word;
    VAR hot : MG.Point (*O*));

This procedure returns the hot spot location (or drawing point) hot for the symbol with the number symbol in the current window's symbol library. The hot spot is given in integer world coordinates.

The hot spot is the point around which the symbol is drawn. See also MG.GetHotSpot.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability

Fonts

CurCharWidth

F CurCharWidth (ch: CHAR): INTEGER;

This function returns the width for the character ch in the current window's font. The width is given in x integer world units. See also MG.CurCharWidth.

SV.Result
MGErr.capDontExist current window has no integer world coordinates capability

If an error occurs, MAXINT is returned.

CurCharBaseline

F CurCharBaseline: INTEGER;

This function returns the height for the base line in the current window's font. The height of the base line is given in y integer world units. See also MG.CurCharBaseLine.

SV.Result
MGErr.capDontExist current window has no integer
If an error occurs, MAXINT is returned.

**CurCharHeight**

\[ F \text{ CurCharHeight}: \text{INTEGER}; \]

This function returns the height for the characters in the current window's font. The height is given in \( y \text{ integer world units} \). See also MG.CurCharHeight.

**SV.Result**

MGErr.capDontExist current window has no integer world coordinates capability

If an error occurs, MAXINT is returned.

**Strings**

**StrWidth**

\[ F \text{ StrWidth(s: String)}: \text{INTEGER}; \]

This function returns the width of the string \( s \) using the current font in the current window. The width is given in \( x \text{ integer world units} \). See also MG.StrWidth.

**SV.Result**

MGErr.capDontExist current window has no integer world coordinates capability

If an error occurs, MAXINT is returned.

\[
\text{MGIW.SetPos}((10000 - \text{MGIW.StrWidth('MENU')}) \ \text{DIV} \ 2, 10000 - \text{MGIW.CurCharHeight});
\]

**DrawStr**

\[ P \text{ DrawStr(s: String)}; \]

This procedure draws the string \( s \) starting in the current integer world posi-
tion and current direction in the current window. See also `MG.DrawLine`.

Updates the current integer world position.

**SV.Result**

**MGErr.capDontExist**

Current window has no integer world coordinates capability.
The unit MGW is used to enhance the capabilities of windows in MG by using floating point world coordinates.

Window coordinates in MG are pixel oriented and integer based and thus device dependent. The lower left corner in a window always has the coordinates (0,0). The x coordinates increase to the right while y coordinates increase upwards.

World coordinates in MGW on the other hand are floating-point based and device independent. World coordinates are individually attached to windows and each window can have its own world coordinates. Both the start value and the direction of world coordinates can be chosen freely.

For example, the x axis can vary between 0.0..100000.0 while the y axis varies between -0.050..+0.050.

World coordinates make it easy to write graphics programs that are independent of the size and resolution of the device. Using world coordinates, data can automatically be scaled when drawing.
MGW versus MGIW

The unit MGIW also supports world coordinates. Compared to MGIW, MGW has the following advantage:

+ greater freedom on selection of world coordinates (MGIW only accepts the integer interval -32768..32767)

At the same time, MGW has the following drawbacks compared to MGIW:

- slower transformations, especially when a math co-processor is absent (80x87 or 80486DX)
- floating point numbers need more space, especially when using polylines, polygons and polybéziers.
Identical Names in MG and MGW

In MGW you will find the equivalent to all those procedures in MG that have parameters for position, distance, direction or viewport, or which affect the current position and direction. The procedures in MGW have the same name and parameter lists as the procedures in MG, but the parameter's data types have been changed from integer to floating-point.

We have therefore shortened the documentation of procedures in MGW that have equivalents in MG. For more information about functions and error codes, we refer you to the documentation for the "sister" procedures in MG.

In order to avoid any misunderstandings using Pascal, it is important to place the units in the right order in the USES list. If you want to use MGW procedures, MGW ought to be placed after MG in the USES list. For example:

```
USES MG,MGW,MGImg;
.. SetPos(0.2,0.9);         (* refers to MGW.SetPos *)
MG.SetPos(100,200);
```

However, by always qualifying procedures with their unit names, confusions can be avoided.

Where maximum speed is crucial, we recommend calling procedures directly in MG or integer based world coordinates in MGW. Moreover, integer coordinates take less space and pixel coordinates entail that the effects of rounding-off can be avoided.

World Coordinates Capability

Before a window can use procedures in MGW, its capabilities must be enhanced using the world coordinates capability. This can be done in one of the following three ways:

1. Individually for each window (screen window coordinates):

   ```
   MG.CreateWin(win1,50,50,200,100);
   MG.EnhanceWin(win1,MGW.cap);
   MG.CreateWin(win2,300,50,200,100);
   MG.EnhanceWin(win2,MGW.cap);
   ```
2. Individually for each window (the screen window has the world coordinates capability):

```pascal
MG.EnhanceWin(MG.CurWin, MGW.cap);
MGW.CreateWin(win1, 50.0, 50.0, 200.0, 100.0);
MGW.CreateWin(win2, 300.0, 50.0, 200.0, 100.0);
```

3. Collectively for all new windows in a screen:

```pascal
MG.EnhanceScr(scr, MGW.cap);
MG.CreateWin(win1, 50, 50, 200, 100);
MG.CreateWin(win2, 300, 50, 200, 100);
```

When the new capability has been installed, the window's world coordinates are set to the range of 0.0..1.0 for both the x and y coordinates. These world coordinates are then said to be *normalized*.

The current position is set to (0.0,0.0), the current direction to 0.0 degrees and the current viewport is assigned to the whole window.

By calling `SetWorld`, new world coordinates can be attached to the window. For instance, if you require that *one* unit in the y direction is of the same physical screen dimension as *one* unit in the x direction (*uniform* world coordinates) this can be accomplished with (the x world is 0.0..1.0):

```pascal
MGW.SetWorld(0.0, 1.0, 0.0, Real(MG.CurWinHeight)/(Real(MG.CurWinWidth)*Real(MG.CurAspect/1000.0)));
```

Uniform coordinates are recommended when transforming point arrays (**MGW.TransformPolyline**) and poly bézier curves (**MGW.TransformPolyBezier**).

## Window Mode and World Mode

A window with the world coordinates capability has, like all windows, a current *window mode* which is defined in **MG**, as well as a window based *world mode* that is defined in **MGW**.

The procedures in **MGW** support the current world mode, but also automatically update the current window mode in **MG**.

Data for the current position, direction and viewport is thus to be found on two levels - on the one hand expressed in world coordinates and degrees in
MGW, and on the other hand expressed in device dependent integer coordinates and deccieges in MG.

For example, if you set a new position using MGW.SetPos and then call MGW.GetPos, it will be found that the device dependent integer position has also been updated.

A word of warning is necessary here; when calling procedures in MG that update the current position, direction or viewport, the equivalent position, direction or viewport in MGW will not be updated. The two coordinate systems are "out of step" with each other (i.e. they are not synchronized!).

If a window has been enhanced with world coordinates, it is important that all calls that affect the current position, direction or viewport are made to procedures in MGW!

For example, MGW.DrawStr updates the current position in MG but not in MGW. MGW.DrawStr updates the current position in both MG and MGW.

As the integer coordinates must be in the interval -32768..32767 it is most likely that you occasionally will submit world coordinates that after transformation would fall outside the integer interval! This results in the error MGW.invalidPos when calling for instance MGW.SetPos.

If you want to know the absolute world limits of the current world coordinates, calling MGW.GetWorldLimits will give you all information about that. By relying on this information, you will also avoid transformation errors (using 8514/A, it is also necessary to use software clipping, as the hardware clipping does not work outside a limited interval - see appendix "MG8514 - 8514/A").

**Concepts and Definitions**

By a window coordinate is meant a coordinate specified in device dependent window coordinates. By a window position is meant a position (x,y) specified in window coordinates.

By a world coordinate is meant a coordinate specified in the current world coordinate system. By a world position is meant a position (x,y) specified in world coordinates.

By a window unit is meant the unit of length (measurement) for window coordinates, normally specified in pixels.
By a *world unit* is meant the unit of length for world coordinates. A world coordinate system can have different world units in the x and y directions.

**Math Co-processor**

You can increase the speed of **MGW** floating point calculations by running your graphics applications on a computer equipped with an 80x87 math co-processor.

All 486-based computers, with the exception of the 486SX's, have a built-in math co-processor.

You may compile applications using world coordinates with the units **MGW** or **MGW87**. In the former case, the compiler will use Borland's Pascal standard 6 bytes **Real**'s, with an accuracy of 11 digits. Using **MGW87**, **Real** is re-declared as 8 bytes **Double**, with 16 digits accuracy.

In both cases, the application has to be compiled with emulation activated (E+ directiv) *if* the program is to be run on a computer *without* a co-processor.

Here are some recommendations about when to use **MGW** and when to use **MGW87**:

**MGW** gives the best possible speed on computers *without* a math co-processor, while **MGW87** gives the best result on computers *with* a math co-processor, but inferior speed on computers without co-processors. **MGW87** also gives better accuracy (not necessary from the viewpoint of *MultiGraphics*, but perhaps in other parts of your application) and uses a standard real format. If real data are stored in a data file, you can be sure that a program compiled with a different compiler will be able to read this data file. This is not the case with Turbo **Real** that has a unique Borland format.

Here are two examples on how speed is greatly affected by the selection of the world unit (executed on a 33 MHz 486DX computer with Borland Pascal 7.0).
1. Calling WorldX

Involves floating point multiplications and divisions.

<table>
<thead>
<tr>
<th>Unit</th>
<th>With 80x87</th>
<th>Without 80x87</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGW</td>
<td>36 000 per sec</td>
<td>17 000 per sec</td>
</tr>
<tr>
<td>MGW87</td>
<td>43 000 per sec</td>
<td>5 000 per sec</td>
</tr>
</tbody>
</table>

2. Calling MovePos

Involves floating point multiplications and calls of the standard functions Sin and Cos.

<table>
<thead>
<tr>
<th>Unit</th>
<th>With 80x87</th>
<th>Without 80x87</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGW</td>
<td>4 000 per sec</td>
<td>2 000 per sec</td>
</tr>
<tr>
<td>MGW87</td>
<td>6 000 per sec</td>
<td>600 per sec</td>
</tr>
</tbody>
</table>

Constants

<table>
<thead>
<tr>
<th>Unit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>unitName</td>
<td>'MGW'</td>
</tr>
<tr>
<td>copyright</td>
<td>'DATABITEN'</td>
</tr>
</tbody>
</table>

Unit name and copyright.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalidWorld</td>
<td>1103</td>
</tr>
<tr>
<td>invalidPos</td>
<td>1104</td>
</tr>
<tr>
<td>invalidAngle</td>
<td>1105</td>
</tr>
<tr>
<td>invalidRect</td>
<td>1106</td>
</tr>
</tbody>
</table>

These are specific error codes for the MGW unit.

nullPoint: MGW.Point = (x:minReal; y:minReal);

This is a special point used to mark a break in polylines and polygons. See DrawPolyline and FillPolygon.
Data Types

**Point** = RECORD \( x, y \): Real; END;

Data types for points using world coordinates.

**Bezier** = ARRAY[0..3] OF MGW.Point;

Data types for Bézier elements using world coordinates.

**WorldMode** = RECORD
  worldx1, worldx2, worldy1, worldy2: Real;
  viewX0, viewY0,
  viewWidth, viewHeight : Real;
  xpos, ypos : Real;
  angle : Real;
END;

Data types for *world mode* for windows that have world coordinates capabilities.

Procedure variables

**EP**: SV.ErrProc;

Local error procedure for the MGW unit.

**revision**: CARDINAL;

Current revision for the implementation of the unit MGW.

**cap**: Capability;

Capability variable for world coordinates. This is used as a parameter to MG.EnhanceWin and MG.EnhanceScr to give windows and screens world coordinates capabilities.
Creating World Windows

**CreateWin**

```pascal
PROCEDURE CreateWin(VAR win : Window;(*O*)
scrX0,scrY0 : Real;
width,height: Real);
```

This procedure creates a window and enhances it with the world coordinates capability. \((x_0,y_0)\) specifies the position for the window's lower left corner, expressed in the screen window's world coordinates. \(\text{width}\) and \(\text{height}\) gives the window's width and the height respectively, expressed in the screen window's world units.

The current screen window must be enhanced prior to this with the world coordinates capability.

**MGW.CreateWin** combines the effect of **MG.CreateWin** and **MG EnhanceWin** with the ability to express the window's position in world coordinates and its size in world units.

A window with the world coordinates capability is removed, just like other windows, by a direct call to **MG.DisposeWin**, or indirectly by calling **MG.DisposeScr** or even **MG.CloseDevice**.

See **MG.CreateWin** and **MG EnhanceWin** for more details.

**SV.Result**

**MGErr.capDontExist** the current window has no world coordinates capability

Errors are also reported in accordance with **MG.CreateWin** and **MG EnhanceWin**.

```pascal
MGW.SetScrWin;
MGW.EnhanceWin(MGW.Cap);
MGW.SetWorld(0.0,1000.0,0.0,1000.0);
MGW.CreateWin(win1,50.0,50.0,200.0,100.0);
MGW.CreateWin(win2,250.0,50.0,200.0,100.0);
```
Window Handles

GetWinHandlesAtPos

P GetWinHandlesAtPos( scrX, scrY: Real;
                     VAR handles : WindowHandleSet (*O*));

This procedure search for and returns in the variable parameter handles, the set of all window handles (0..254) in the current screen whose corresponding windows enclose the given position (scrX, scrY). The position is given in screen world coordinates. If this position is not located inside any window of the screen, the empty set is returned.

SV.Result
MGErr.processWithoutWin       the calling process has no window
MGErr.capDontExist            current window has no world
                                coordinates capability

When an error occurs, the empty set is returned.

The Current World Mode

World Mode

GetWorldMode

P GetWorldMode(VAR mode: WorldMode (*O*));

Returns the current window's world mode in the variable mode.

The data type WorldMode is defined above under "Data Types".

This procedure is normally used with SetWorldMode to save the current world mode and then restore it.

SV.Result
MGErr.processWithoutWin       the calling process has no window
MGErr.capDontExist            current window has no world
                                coordinates capability
VAR
  win         : MG.Window;
  oldWorldMode: MGW.WorldMode;
  winMode     : MG.WindowMode;
  ..
  MG.EnhanceWin(win,MGW.cap);
  MGW.\texttt{GetWorldMode}(oldWorldMode);
  MG.GetWinMode(winMode);
 ..
  MG.SetWinMode(&oldWinMode);
  MGW.SetWorldMode(oldWorldMode);

\texttt{SetWorldMode}  

\texttt{P SetWorldMode(VAR \textit{mode}: WorldMode (*I*))};

This procedure sets the current window's world mode. It also updates the current window mode in \texttt{MG} with regard to position, viewport and direction.

\texttt{SetWorldMode does not verify that the information in the variable \textit{mode} is correct. The programmer must check this himself.}

The procedure is normally used with \texttt{GetWorldMode} in order to store and restore a particular world mode.

\texttt{SV.Result}
\begin{itemize}
  \item \texttt{MGErr.processWithoutWin} \quad \text{the calling process has no window}
  \item \texttt{MGErr.capDontExist} \quad \text{current window has no world}
      \quad \text{coordinates capability}
\end{itemize}

\texttt{World Coordinates}

\texttt{SetWorld}  

\texttt{P SetWorld(x1,x2,y1,y2: Real);} 

This procedure defines the current world coordinate system for the current window.

\texttt{x1} specifies the world coordinate for the window's left hand side, while \texttt{x2} specifies the world coordinate for the window's right hand side. \texttt{x1 <> x2}. Note that \texttt{x2 < x1} is allowed.
y1 specifies the world coordinate for the window's lower side, while y2 specifies the world coordinate for the window's top side. y1 <> y2. Note that y2 < y1 is allowed.

All subsequent calls by procedures in MGW will use the new the world coordinates.

The current window must have the world coordinates capability.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no world coordinates capability
invalidWorld invalid world coordinates

MG.SetWin(win);
MGW.SetWorld((-5000.0,10000.0,5.0,-5.0));

GetWorld

P GetWorld(VAR x1,x2,y1,y2: Real (*O*));

This procedure returns the current window's world coordinate system.

x1 specifies the world coordinate for the window's left hand side, while x2 specifies the world coordinate for the window's right hand side.

y1 specifies the world coordinate for the window's lower side, while y2 specifies the world coordinate for the window's top side.

The current window must have the world coordinates capability.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no world coordinates capability

GetWorldLimits

P GetWorldLimits(VAR xmin,xmax,ymin,ymax: Real (*O*));

This procedure returns the absolute limits for the current window's world coordinates.
**MGW - World Coordinates**

`xmin` specifies the smallest acceptable world coordinate for x values, while `xmax` specifies the largest acceptable world coordinate for x values.

`ymin` specifies the smallest acceptable world coordinate for y values, while `ymax` specifies the largest acceptable world coordinate for y values.

If a drawing operation specifies a world coordinate outside these limits, the result will be an error during transformation to device coordinates.

The current window must have the world coordinates capability.

**SV.Result**

- `MGErr.processWithoutWin` the calling process has no window
- `MGErr.capDontExist` current window has no world coordinates capability

### Converting Coordinates

#### DevX

```pascal
F DevX(x: Real): INTEGER;
```

This function converts the *world coordinate* `x` in the current window to a device specific window coordinate.

The current window must have the world coordinates capability.

**SV.Result**

- `MGErr.processWithoutWin` the calling process has no window
- `MGErr.capDontExist` current window has no world coordinates capability

If an error occurs, `MAXINT` is returned.

```pascal
VAR
  x: INTEGER;
  ..
  x := MGW.DevX(2125.0);
```
**DevY**

\[
\text{F DevY}(y: \text{Real}): \text{INTEGER};
\]

This function converts the world coordinate \( y \) in the current window to a device specific window coordinate.

The current window must have the world coordinates capability.

**SV.Result**
- \( \text{MGErr.processWithoutWin} \): the calling process has no window
- \( \text{MGErr.capDontExist} \): current window has no world coordinates capability

If an error occurs, \( \text{MAXINT} \) is returned.

**DevWidth**

\[
\text{F DevWidth}(w: \text{Real}): \text{INTEGER};
\]

This function converts the width \( w \) in world units to a device specific width in window units.

The current window must have the world coordinates capability.

**SV.Result**
- \( \text{MGErr.processWithoutWin} \): the calling process has no window
- \( \text{MGErr.capDontExist} \): current window has no world coordinates capability

If an error occurs, \( \text{MAXINT} \) is returned.

```pascal
VAR
  w: INTEGER;
  ..
  w := \text{MGW.DevWidth}(1000.0);
```

**DevHeight**

\[
\text{F DevHeight}(h: \text{Real}): \text{INTEGER};
\]

This function converts the height \( h \) in world units to a device specific height in window units.
The current window must have the world coordinates capability.

**SV.Result**

- **MGErr.processWithoutWin**: the calling process has no window
- **MGErr.capDontExist**: current window has no world coordinates capability

If an error occurs, **MAXINT** is returned.

**WorldX**

\[ F \text{ WorldX} (x: \text{ INTEGER}) : \text{ Real}; \]

This function converts the window coordinate \( x \) to a world coordinate.

The current window must have the world coordinates capability.

**SV.Result**

- **MGErr.processWithoutWin**: the calling process has no window
- **MGErr.capDontExist**: current window has no world coordinates capability

If an error occurs, **maxReal** is returned.

\[ \text{MGW.SetPos(MGW.WORLDX(100),MGW.WORLDY(100))}; \]

**WorldY**

\[ F \text{ WorldY} (y: \text{ INTEGER}) : \text{ Real}; \]

This function converts the window coordinate \( y \) to a world coordinate.

The current window must have the world coordinates capability.

**SV.Result**

- **MGErr.processWithoutWin**: the calling process has no window
- **MGErr.capDontExist**: current window has no world coordinates capability

If an error occurs, **maxReal** is returned.
WorldWidth

\[ F \text{ WorldWidth}(w: \text{ INTEGER}): \text{ Real}; \]

This function converts the width \( w \) in window units to a width in world units.

The current window must have the world coordinates capability.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no world
coordinates capability

If an error occurs, maxReal is returned.

\[ \text{MGW.SetPos}(0.1 + \text{ MGW.WorldWidth}(100), 0.5); \]

WorldHeight

\[ F \text{ WorldHeight}(h: \text{ INTEGER}): \text{ Real}; \]

This function converts the height \( h \) in window units to a height in world units.

The current window must have the world coordinates capability.

SV.Result
MGErr.processWithoutWin the calling process has no window
MGErr.capDontExist current window has no world
coordinates capability

If an error occurs, maxReal is returned.

Position

SetPos

\[ P \text{ SetPos}(x, y: \text{ Real}); \]

This procedure sets the new current position in the current window. \( x \) and \( y \) are specified in world coordinates. See also MG.SetPos.
SV.Result
MGErr.capDontExist current window has no world
coordinates capability
invalidPos invalid position
MGW.SetPos(0.5,0.6);

SetRelPos

P SetRelPos(dx,dy: Real);
This procedure sets the new position relative to the current position in the
current window. \(dx\) and \(dy\) are specified in world units. See also
MGW.SetRelPos.

SV.Result
MGErr.capDontExist current window has no world
coordinates capability
invalidPos invalid position
MGW.SetRelPos(0.1,0.1);

MovePos

P MovePos(dist: Real);
This procedure moves the current position the distance \(dist\) in the current
direction. \(dist\) is specified in \(x\) world units. See also MGW.MovePos.

SV.Result
MGErr.capDontExist current window has no world
coordinates capability
invalidPos invalid position
MGW.SetDir(45.0);
MGW.MovePos(0.5);

GetPos

P GetPos(VAR x,y: Real (*O*));
This procedure reads the current position in the current window. \(x\) and \(y\) are
expressed in world coordinates. See also \texttt{MG.GetPos}.

\begin{verbatim}
 SV.Result
 MGErr.capDontExist current window has no world
coordinates capability
\end{verbatim}

### CurX

\begin{verbatim}
 F CurX: Real;

This function returns the current x position in the current window. The
position is expressed in x world coordinates. See also \texttt{MG.CurX}.

SV.Result
 MGErr.capDontExist current window has no world
coordinates capability

If an error occurs, \texttt{maxReal} is returned.

VAR
  x: Real;
  ...
  x := MGW.CurX;
\end{verbatim}

### CurY

\begin{verbatim}
 F CurY: Real;

This function returns the current y position in the current window. The
position is expressed in y world coordinates. See also \texttt{MG.CurY}.

SV.Result
 MGErr.capDontExist current window has no world
coordinates capability

If an error occurs, \texttt{maxReal} is returned.

### GetFastXY

\begin{verbatim}
 P GetFastXY(VAR x,y: Real (*IO*));

Based on the given \texttt{x} and \texttt{y} values, this procedure returns proposals for the
nearest \texttt{x} and \texttt{y} values which would result in a higher drawing speed. \texttt{x} and
\end{verbatim}
y are specified in world coordinates. See also \texttt{MG.GetFastXY}.

\textbf{SV.Result}
\texttt{MGErr.capDontExist} \hspace{1cm} \text{current window has no world coordinates capability}

\textbf{VAR}
\begin{verbatim}
x, y: Real;
\end{verbatim}

\begin{verbatim}
x := 0.3; y := 0.8;
MGW.GetFastXY(x, y);
MGW.SetPos(x, y);
\end{verbatim}

\textbf{Direction}

\begin{tabular}{|l|}
\hline
\textbf{SetDir} & \textbf{DL} \\
\hline
\texttt{P SetDir(angle: Real);} & \\
\hline
\end{tabular}

This procedure sets the drawing direction in the current window. \texttt{angle} is specified in degrees and must be within the range of 0.0..360.0. See also \texttt{MG.SetDir}.

\textbf{SV.Result}
\texttt{MGErr.capDontExist} \hspace{1cm} \text{current window has no world coordinates capability}

\texttt{invalidAngle} \hspace{1cm} \text{invalid angle}

\begin{verbatim}
MGW.SetDir(45.0);
\end{verbatim}
**TurnDir**

P TurnDir(angle: Real);

Turns the current window's drawing direction using the angle in the parameter `angle`. The angle is specified in *degrees*. See also `MG.TurnDir`.

**SV.Result**

MGErr.capDontExist  current window has no world coordinates capability

invalidAngle  invalid angle

MGW.TurnDir(-30.0);

**CurDir**

F CurDir: Real;

This function returns the current drawing direction in the current window. The direction is expressed in degrees. See also `MG.CurDir`.

**SV.Result**

MGErr.capDontExist  current window has no world coordinates capability

If an error occurs, `maxReal` is returned.

MGW.SetDir(MGW.CurDir/2.0);

**Viewport**

**SetView**

P SetView(x0,y0,width,height: Real);

This procedure sets the current window's viewport. x0 and y0 are specified in world coordinates, while `width` and `height` are given in world units. See also `MG.SetView`.

When a window capability has been installed, the viewport is set to the whole window.
MGW - World Coordinates

**GetView**

```pascal
P GetView(VAR x0,y0,width,height: Real (*O*));
```

This procedure reads the location and size of the current window's viewport. `x0` and `y0` are expressed in world coordinates while `width` and `height` are given in world units. See also **MG.GetView**.

SV.Result
MGErr.capDontExist current window has no world coordinates capability

**Draw Graphics**

**Points**

```pascal
P DrawPoint (x,y : Real;
           color: MG.DeviceColor);
```

This procedure draws a point at world position `(x,y)` with device color `color`, using the current color mask and drawing mode. If the position lies outside the current viewport, nothing is drawn. The current position and colors are not modified.

See also **MG.DrawPoint**.
PointDeviceColor

F PointDeviceColor(x, y: Real): DeviceColor;

This function returns the device color for a point (pixel) at world position (x, y) in the current window. The specified position must be within the current viewport.

See also MG.PointDeviceColor.

SV.Result
MGErr.capDontExist current window has no world coordinates capability
invalidPos invalid position
MGErr.invalidDeviceColor invalid device color

Straight Lines

DrawLineTo

P DrawLineTo(x, y: Real);

This procedure draws a straight line from the current position to (x, y) in the current window. x and y are specified in world coordinates. (x, y) becomes the new current world position. See also MG.DrawLineTo.

SV.Result
MGErr.capDontExist current window has no world coordinates capability
invalidPos invalid final position
MGW.SetPos(-20.0, 150.0);
MGW.DrawLineTo(-5.0, 200.0);
**DrawLineRel**

```pascal
P DrawLineRel(dx, dy: Real);
```

This procedure draws a straight line relatively from current position \((x0,y0)\) to \((x0+dx, y0+dy)\). The displacement is given in world units. \((x0+dx, y0+dy)\) becomes the new current position. See also `MG.DrawLineRel`.

**SV.Result**
- `MGErr.capDontExist`: current window has no world coordinates capability
- `invalidPos`: invalid final position

```pascal
MGW.DrawLineRel(0.1, -0.1);
```

**DrawLine**

```pascal
P DrawLine(dist: Real);
```

This procedure draws a straight line using the length `dist` from the current position and in the current drawing direction. The distance is given in world units. See also `MG.DrawLine`.

**SV.Result**
- `MGErr.capDontExist`: current window has no world coordinates capability
- `invalidPos`: invalid final position

```pascal
MGW.SetDir(135.0);
MGW.DrawLine(0.500);
```
Polyline functions

## TransformPolyline

```pascal
P TransformPolyline(VAR polyline; (* ARRAY OF MGW.Point*)
  first, last   : Word;
  scaleX, scaleY: Real;
  angle        : Real;
  shearX, shearY: Real);
```

Transforms the points with index `first` to `last` (`last >= first`) in the point array `polyline`. The points are scaled in the x direction by the factor `scaleX` (1.0 = unscaled) and in the y direction by `scaleY` (1.0 = unscaled). The points are rotated by `angle` degrees (counter clockwise if > 0.0, clockwise if < 0.0). The points are sheared in the x direction by `shearX` (0.0 = unsheared) and in the y direction by `shearY` (0.0 = unsheared). Points are specified in world units.

A special point, `MGW.nullPoint`, is used to separate different polylines in the array of points from each other. `MGW.nullPoint` is not transformed.

See also `MG.TransformPolyline`.

If you want to preserve the proportions of the polyline under rotation, it is recommended to use uniform world coordinates (see "World Coordinates Capability" above).

See `MG.TransformPolyline`.

```pascal
SV.Result
MGErr.invalidIndex last >= first!
```

```pascal
pa: ARRAY[0..999] OF MGW.Point;
MGW.TransformPolyline(pa, 0, 299, 2.0, 2.0, 90.0, 0.0, 0.0);
MGW.DrawPolyline(pa, 0, 299);
```

## DrawPolyline

```pascal
P DrawPolyline(VAR polyline; (*I*)
  first, last: Word);
```

This procedure draws a `polyline` based on the current position and the array of relative points that are given in the variable parameter `polyline`. The
points are specified in world units. The current world position is set relative to the last point used in the array \((\text{current position} + \text{polyline[last]})\).

A special point, \texttt{MGW.nullPoint}, is used to separate different polylines in the array of points from each other.

See also \texttt{MG.DrawPolyline}.

<table>
<thead>
<tr>
<th>SV.Result</th>
<th>MGErr.capDontExist</th>
<th>current window has no world coordinates capability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MGErr.tooManyPoints</td>
<td>too many points</td>
</tr>
</tbody>
</table>

Rectangles

\textbf{DrawBox} \quad DL

\begin{verbatim}
P DrawBox(width, height: Real);
\end{verbatim}

This procedure draws a rectangle (box) with sides \texttt{width} and \texttt{height}, with the starting corner in the current window position and the base in the \textit{x direction}. The width and height are specified in \textit{x world units}, respectively. The current world position remains unchanged. See also \texttt{MG.DrawBox}.

<table>
<thead>
<tr>
<th>SV.Result</th>
<th>MGErr.capDontExist</th>
<th>current window has no world coordinates capability</th>
</tr>
</thead>
</table>

\texttt{MGW.DrawBox(0.3,0.1)};

\textbf{DrawRect} \quad DL

\begin{verbatim}
P DrawRect(width, height: Real);
\end{verbatim}

This procedure draws a rectangle with sides \texttt{width} and \texttt{height}, with the starting corner in the current window position and the base in the \textit{current drawing direction}. The lengths of the sides are specified in \textit{x world units}. The current world position remains unchanged. See also \texttt{MG.DrawBox}.
Circles, Arcs and Circle Sectors

**DrawCircle**

```pascal
P DrawCircle(radius: Real);
```

This procedure draws a *circle* with its center in the current world position using the radius in the parameter `radius`. The radius is specified in *x world units*. See also MG.DrawCircle.

**DrawArc**

```pascal
P DrawArc(radius, centralAngle: Real);
```

This procedure draws an *arc* with its center in the current world position, the starting angle in the current drawing direction, the radius `radius` and the central angle `centralAngle`. The radius is specified in *x world units* and the central angle in *degrees*. The current world position is not changed. The current drawing direction is assigned to the arc's final angle. See also MG.DrawArc.
**DrawSector**

\[
P \text{DrawSector}(radius, \text{centralAngle}: \text{Real});
\]

This procedure draws a circle sector with the center in the current world position, the starting angle in the current drawing direction, the radius \(radius\) and the central angle \(\text{centralAngle}\). The radius is specified in x world units and the central angle in degrees. The current position is not changed. The current direction is assigned to the sector's final angle. See also MG.DrawSector.

**SV.Result**

MGErr.capDontExist  
current window has no world coordinates capability

MGErr.invalidRadiusOrAxis  
invalid radius

invalidAngle  
invalid angle

\[
\text{MGW.DrawSector}(0.5,45.0);
\]

**Ellipses, Elliptical Arcs and Sectors of Ellipses**

**DrawEllipse**

\[
P \text{DrawEllipse}(a,b,\text{majorAxisAngle}: \text{Real});
\]

This procedure draws an ellipse with its center in the current world position, the direction of the major axis in accordance with \(\text{majorAxisAngle}\), the length of the major axis \(a\) and the length of the minor axis \(b\). The major axis and minor axis are specified in x world units. The direction of the major axis is specified in degrees. The current world position and direction are not changed. See also MG.DrawEllipse.

**SV.Result**

MGErr.capDontExist  
current window has no world coordinates capability

MGErr.invalidRadiusOrAxis  
invalid axes

invalidAngle  
invalid angle

\[
\text{MGW.DrawEllipse}(0.5,0.2,90.0);
\]
DrawEllipseArc

\[
P \text{DrawEllipseArc}(a, b, \text{majorAxisAngle} : \text{Real}; \\
\quad \text{centralAngle} \quad : \text{Real});
\]

This procedure draws an *elliptical arc* with its center in the current world position, the starting angle in the current drawing direction, the major axis \(a\), the minor axis \(b\), the direction of the major axis \text{majorAxisAngle} and the central angle \text{centralAngle}. The major and minor axes are specified in \(x\) world units. The directions of the major axis and the central angle are specified in degrees. The current world position is not changed. The current direction is assigned to the arc's final angle. See also MG.DrawEllipseArc.

SV.Result
MGErr.capDontExist \quad \text{current window has no world coordinates capability}
MGErr.invalidRadiusOrAxis \quad \text{invalid axes}
invalidAngle \quad \text{invalid angle}

\[
\text{MGW.DrawEllipseArc}(3500.0, 2200.0, 30.0, -20.0);
\]

DrawEllipseSector

\[
P \text{DrawEllipseSector}(a, b, \text{majorAxisAngle} : \text{Real}; \\
\quad \text{centralAngle} \quad : \text{Real});
\]

This procedure draws a *sector of an ellipse* with its center in the current world position, the starting angle in the current drawing direction, the major axis \(a\), the minor axis \(b\), the direction of the major axis \text{majorAxisAngle} and the central angle \text{centralAngle}. The major and minor axes are specified in \(x\) world units. The directions of the major axis and the central angle are specified in degrees. The current world position is not changed. The current direction is assigned to the sector's final angle. See also MG.DrawEllipseSector.

SV.Result
MGErr.capDontExist \quad \text{current window has no world coordinates capability}
MGErr.invalidRadiusOrAxis \quad \text{invalid axes}
invalidAngle \quad \text{invalid angle}

\[
\text{MGW.DrawEllipseSector}(3500.0, 2200.0, 30.0, -20.0);
\]
Bézier Curves

TransformPolyBezier

P TransformPolyBezier(VAR polyline; ARRAY OF MGW.Bezier)  
  first, last   : Word;  
  scaleX, scaleY: Real;  
  angle        : Real;  
  shearX, shearY: Real);

Transforms the Bézier elements with index first to last (last >= first) in the Bézier array polyBez. The Bézier points are scaled in the x direction by the factor scaleX (1.0 = unscaled) and in the y direction by scaleY (1.0 = unscaled). The points are rotated by angle degrees (counter clockwise if > 0.0, clockwise if < 0.0). The points are sheared in the x direction by shearX (0.0 = unsheared) and in the y direction by shearY (0.0 = unsheared). Points are specified in world units.

See also MG.TransformPolyBezier.

If you want to preserve the proportions of the polybézier under rotation, it is recommended to use uniform world coordinates (see "World Coordinates Capability" above).

SV.Result
MGErr.invalidIndex  
last >= first!

pb: ARRAY[0..499] OF MGW.Bezier;  
..  
MGW.TransformPolyBezier(pb,100,199,0.5,0.5,45.0,0.0,0.0);  
MGW.DrawPolyBezier(pa,100,199);

DrawPolyBezier

P DrawPolyBezier(VAR polyBez; (* ARRAY OF Bezier*) (*I*))  
  first, last: Word);

Draws a polybézier curve relative the current position and the array of relative Bézier elements, of the type MGW.Bezier, found in the variable parameter polyBez. The polybézier starts from the Bézier element with the index first and ends with the element with the index last, where last >= first.

The control points of the Bézier curve is given in world units. The new cur-
rent position is the point (relative to the current start position) which is
given as the last control point of the Bézier element with index \texttt{last}.

\textbf{SV.Result}

\textbf{MGErr.capDontExist} \quad \text{current window has no world}
coordinates capability

\textbf{outOfMem} \quad \text{can't create internal temporary}
conversion buffer

\textbf{tooManyPoints} \quad \text{internal buffer is filled up}

Patterns

\textbf{FillRect} \quad \text{DL}

\texttt{P FillRect(x0,y0,width,height: Real);}

This procedure fills the specified rectangular area in the current window
using the current pattern. \((x0,y0)\) specifies the rectangle's lower left corner,
expressed in world coordinates. \texttt{width} and \texttt{height} specify the rectangle's
width and the height in world units. See also \texttt{MG.FillRect}.

\textbf{SV.Result}

\textbf{MGErr.capDontExist} \quad \text{current window has no world}
coordinates capability

\textbf{MGErr.invalidRect} \quad \text{invalid rectangle}

\texttt{MGPat.SetGenPattern(MGPat.diagUp);}
\texttt{MGW.FillRect(-0.4,0.2,2.0,1.0);}

\textbf{FillPolygon} \quad \text{DL}

\texttt{P FillPolygon(VAR vertices; (*I*)}
\texttt{first,last: Word);} \texttt{)}

This procedure fills a \textit{polygon}, based on the current position and the array of
\textit{relative} point data that is given in the variable parameter \texttt{vertices}. The
polygon starts at the point with the index \texttt{first} and ends at the point with the
index \texttt{last}, where \texttt{last} >= \texttt{first}. Points are specified in world units.

A special point, \texttt{MGW.nullPoint}, is used to fill separate polygons and to
prevent the inner parts of a polygon from being filled.
FillCircle

P FillCircle(radius: Real);

This procedure fills a circle with the current pattern. The circle's center is at the current position and the radius is \(\text{radius}\). The radius is specified in \(x\) world units. The current position is not changed. See also MG.FillCircle.

FillSector

P FillSector(radius, centralAngle: Real);

This procedure fills a circle sector with the current pattern. The center is at the current window position, the starting angle in the current drawing direction, the radius is \(\text{radius}\) and the central angle \(\text{centralAngle}\). The radius is specified in \(x\) world units. The central angle is specified in degrees. The current position and direction are not changed. See also MG.FillSector.
FillEllipse DL

P FillEllipse(a,b,majorAxisAngle: Real);

This procedure fills an ellipse with the current pattern. The ellipse's center is at the current world position. a specifies the ellipse's major axis and b the minor axis. majorAxisAngle gives the direction of the major axis. The major and minor axes are specified in x world units. The direction of the major axis is specified in degrees. The current position and direction are not changed. See also MG.FillEllipse.

SV.Result
MGErr.capDontExist current window has no world coordinates capability
MGErr.invalidRadiusOrAxis invalid axes
invalidAngle invalid angle

MG.MatchForeground(MG.red);
MG.MatchBackground(MG.blue);
MGPat.SetPattern(MGPat.halftone2);
MGW.FillEllipse(10.0,15.0,15.0);

FillEllipseSector DL

P FillEllipseSector(a,b,majorAxisAngle: Real;
    centralAngle : Real);

This procedure fills a sector of an ellipse with the current pattern. The ellipse's center lies in the current window position, the sector's starting angle in the current drawing direction, the major axis is given in a, the minor axis in b, the direction of the major axis in majorAxisAngle and the sector's central angle in centralAngle. The major and minor axes are specified in x world units. The directions of the major axis and the central angle are specified in degrees. The current position and direction are not changed. See also MG.FillEllipseSector.
Filling of Polybézier Curves

**FillPolyBezier**

```pascal
P FillPolyBezier(VAR polyBez; (* ARRAY OF Bezier*) (*I*)
               first,last: Word);
```

Fill a polybézier curve relative the current position and the array of relative Bézier elements, of the type `MGW.Bezier`, found in the variable parameter `polyBez`. The polybézier starts from the Bézier element with the index `first` and ends with the element with the index `last`, where `last >= first`.

*The border line is also drawn (in contrast to `MGW.FillPolygon`), if the line style is not set to `MG.nullLineStyle`.*

The current position and drawing direction are not changed.

The only permitted start index of the polybézier array is 0. The coordinates of the control points do not specify absolute coordinates, instead these are given *relative to the current world position*. 

**SV.Result**

- `MGErr.capDontExist` current window has no world coordinates capability
- `MGErr.invalidRadiusOrAxis` invalid axes
- `invalidAngle` invalid angle

**Outcomes**

- `SV.Result` current window has no world coordinates capability
- `MGErr.capDontExist` current window has no world coordinates capability
- `outOfMem` can't create internal temporary conversion buffer
- `tooManyPoints` internal buffer is filled up
Ref 22-34  

MGW - World Coordinates

Copying

**CopyRect**

```
P CopyRect(source      : Window;
            x0,y0       : Real;
            width,height: Real);
```

This procedure copies the graphics in a rectangular area, located in the window `source`, to the current window. The original rectangle's lower left corner is specified using \((x0,y0)\) in `source` world coordinates. The width and height are specified using `width` and `height` in `source` world units. The copy's lower left corner is placed at the current position and window. See also `MG.CopyRect`.

**SV.Result**

MGErr.capDontExist current window has no world coordinates capability

MG.SetWin(win2);
MGW.SetPos(0.0,0.0);
MGW.CopyRect(win1,0.0,0.0,1.0,1.0);

Symbols

**CurSymbolWidth**

```
P CurSymbolWidth(symbol: Word): Real;
```

This procedure returns the width for the symbol with the number `symbol` in the current window's symbol library. The width is given in \(x\) world units. See also `MG.CurSymbolWidth`.

**SV.Result**

MGErr.capDontExist current window has no world coordinates capability

If an error occurs, `maxReal` is returned.
CurSymbolHeight

P CurSymbolHeight(symbol: Word): Real;

This procedure returns the height for the symbol with the number symbol in current window's symbol library. The height is given in y world units. See also MG.CurSymbolHeight.

SV.Result
MGErr.capDontExist current window has no world coordinates capability

If an error occurs, maxReal is returned.

GetHotSpot

P GetHotSpot( symbol: Word;
VAR hot : MGW.Point (*O*));

This procedure returns the hot spot location (or drawing point) hot for the symbol with the number symbol in the current window's symbol library. The hot spot is given in world coordinates.

The hot spot is the point around which the symbol is drawn. See also MG.GetHotSpot.

SV.Result
MGErr.capDontExist current window has no world coordinates capability

Fonts

CurCharWidth

F CurCharWidth(ch: CHAR): Real;

This function returns the width for the character ch in the current window's font. The width is given in x world units. See also MG.CurCharWidth.
CurCharBaseline

F CurCharBaseline: Real;

This function returns the height for the base line in the current window's font. The height of the base line is given in y world units. See also MG.CurCharBaseLine.

Strings

StrWidth

F StrWidth(s: String): Real;

This function returns the width of the string s using the current font in the current window. The width is given in x world units. See also MG.StrWidth.
If an error occurs, **maxReal** is returned.

```
MGW.SetPos((1.0 - MGW.StrWidth('MENU'))/2.0,
           1.0 - MGW.CurCharHeight);
```

**DrawStr**

```
procedure DrawStr(s: String);
```

This procedure draws the string `s` starting in the current world position and current direction in the current window. See also **MG.DrawStr**.

Updates the current world position.

**SV.Result**

`MGErr.capDontExist` current window has no world coordinates capability
The unit MGIO is used to enhance the capability of windows in MG with character based input and output, or text i/o for short.

MGIO includes support to:

- set the current drawing position expressed in text based column and row units,
- tabulate,
- clear windows and lines,
- format and print characters, strings, integer and floating-point numbers,
- automatically start a new text line when text output reach the window's right hand side and scroll text upwards at the bottom of the window,
- write and read text with character attributes such as bold, shadow, engrave, emboss and underline,
- define the text input cursor,
- read the keyboard, including function keys and special keys,
- edit and read characters, strings, integer and floating-point numbers,
- give proposals on text input,
- specify termination keys for text input,
- write your own input procedures (e.g., mouse based),
- manage multiple string editors concurrently,
- verify and modify input character by character,
- create delays etc.

The i/o functions are installed as a window capability in windows that require text handling. The text i/o capability can be combined with other capabilities, e.g. world coordinates.
Text I/O Capability

Before a window can use procedures in MGIO, its capabilities must be enhanced with the text i/o capability. This can be done by one of the following three methods:

1. Individually for each window using the following MG procedures:

   MG.CreateWin(win1,50,50,200,100);
   MG.EnhanceWin(win1,MGIO.cap);

   MG.CreateWin(win2,300,50,200,100);
   MG.EnhanceWin(win2,MGIO.cap);

2. Individually for each window using the MGIO procedure:

   MGIO.CreateWin(win1,50,50,200,100);
   MGIO.CreateWin(win2,300,50,200,100);

3. Collectively for all new windows in a screen using:

   MG.EnhanceScr(scr,MGIO.cap);
   MG.CreateWin(win1,50,50,200,100);
   MG.CreateWin(win2,300,50,200,100);

   When the new capability has been installed, the current position is set to column 1 and row 1, expressed in the current font's average width and height.

Window Mode and Text Mode

A window with text i/o capability has, just like all other windows, a current window mode defined in MG, as well as a window based text i/o mode, which is defined in MGIO. The contents of this i/o mode is defined by the data type IOMode.

The procedures in MGIO take into consideration and up-date both the current i/o mode and the current window mode.

Font transformations in MG affect text handling in MGIO.
Text Position and Drawing Position

The current text position is identical with the current drawing position in MG. The position can be set using procedures in MG, MGIW or MGW, as well as by using column and row oriented procedures in MGIO.

Columns and rows are numbered from 1 upwards. Row 1 is at the top of the window.

The row height is of the same size as the current font's character height, while the column width is of the same size as the average width of the current font's characters. Information about the average width is stored in the font file.

Character Attributes

All characters written or read can be enhanced with character attributes from the enumerated type CharacterAttributes. The current attributes are set by calling SetCharAttributes and can be read by calling GetCharAttributes. Each window has its own bunch of character attributes. However, the character attributes are only applied if the current drawing mode is replace or stamp.

Here are the supported attributes:

- **underline** underlining
- **bold** characters with shadows
- **shadow** "etched", engraved characters
- **engrave** "elevated" characters
- **emboss**

You can combine the attributes underline and bold freely with each other and all other attributes, whereas shadow, engrave and emboss can't be used at the same time.

The attributes bold..emboss increase the character widths and shadow..emboss also increase the character height of the current font during character output. You can call CurAttrCharWidth, CurAttrCharHeight, CurAttrCharBaseline and AttrStrWidth to get the current dimensions.

You can change the position of shadows and enlightened parts of the attributes
shadow, engrave and emboss by calling the procedure SetCharAttributePos. As a default, characters are enlightened by a "light source" placed above and to the left of the text and therefore shadows are falling below and to the right. As a default, the shadows are displaced with 1/500 of the current screen width for shadow and with 1/1000 of the current screen width for engrave and emboss - however never less than 1 pixel.

As a default, dark gray (MG.darkGray) is used for shadow effects with the attributes shadow and emboss and white (MG.white) is used to display illuminated parts with the attributes engrave and emboss. Here are a few recommendations for applicable color combinations (use MG.SetForeground, MG.MatchBackground, etc. to set them):

<table>
<thead>
<tr>
<th>Attribute</th>
<th>background color</th>
<th>foreground color</th>
</tr>
</thead>
<tbody>
<tr>
<td>shadow</td>
<td>moderately bright</td>
<td>bright</td>
</tr>
<tr>
<td>engrave</td>
<td>MG.lightGray</td>
<td>MG.black</td>
</tr>
<tr>
<td>emboss</td>
<td>MG.lightGray</td>
<td>MG.lightGray</td>
</tr>
</tbody>
</table>

If desirable, the attribute colors can be changed in the current window by calling SetCharAttributeColors. You can read the current attribute colors with GetCharAttributeColors.

For compatibility reasons with older MultiGraphics versions, you can also set and read the underline attribute using SetUnderline and CurUnderline, respectively. However we recommend using the more general procedures SetCharAttributes and GetCharAttributes.

Text Output

MGIO includes output procedures for new lines (WrLn), characters (WrChar), character strings (WrStr), integers (WrInt, WrWord and WrLong) as well as floating-point numbers (WrReal and WrFixReal).

Scrolling

Text strings that are written using MG.DrawStr are clipped against the sides of the window, or against the current viewport if this is smaller. The new current position is not affected by this clipping, but is instead the same as if the complete string had been written.

The character output procedures in MGIO act in the same way as DrawStr,
if scrolling is switched off.

If, on the other hand, scroll mode is on (the default), new lines are automatically created when the text output reaches the window's right hand side, and the contents of the window is automatically moved upwards (scrolled) when the text output reaches the bottom of the window.

N.B. new lines and scrolling are carried out at the window's sides, and not at the viewport's sides.

In order to achieve the maximum printing speed, it is recommended that scrolling be switched off and that characters are written using "fast positions" (see MG.GetFastXY, MGIW.GetFastXY and MGW.GetFastXY).

Redirection of Standard I/O

The standard procedures Write, WriteLn, Read and ReadLn in Pascal are redirected automatically to use graphics oriented routines in MGIO as soon as at least one open window has the text i/o capability.

The Write and Read procedures now run in graphics mode, using the current window mode and text mode, i.e. the current font, drawing mode, foreground and background colors, color mask, character attributes, cursor type etc..

Those windows that use standard procedures must still be given text i/o capability.

The standard procedures are reset automatically to text mode (System, CRT, etc.) when the last window which has the text i/o capability has been closed.

If you want to use the unit CRT in text mode, it is important that you place the CRT unit before the MGIO unit in the USES list.

USES CRT, MG, MGIO;
MG.EnhanceScr(MG.CurScr,MGIO.cap);
MG.CreateWin(win,50,50,200,100);
WriteLn('Name: 's:20,' Age: ',i:5);
Write('Give a floating-point number: '); ReadLn(r);
Input

MGIO includes procedures to read characters (RdChar), strings (RdStr), integers (RdInt, RdWord and RdLong) as well as floating-point numbers (RdReal and RdFixReal).

The input procedures themselves verify that the maximum permitted number of characters is not exceeded, that only permitted characters are used and that characters are written in the positions allowed. When invalid input occurs, data is ignored and a warning is sounded (if this alarm has not been switched off).

When reading floating-point numbers for example, the following are accepted numbers; one decimal point, one 'E' or 'e' as well as plus or minus signs in the first position as well as after any possible exponent characters.

The input procedures allow line editing using back spacing and deleting, setting the current position in the edited string, overwriting and inserting new characters etc. Normally, editing is controlled via the keyboard using backspace, arrows and deleting keys.

Input is terminated using a key or combination of keys from the set of accepted termination keys in the current window. Normally, Enter, Ctrl-C, Ctrl-Z and Esc are used as termination keys, but this can be changed using SetTermKeys. Perhaps you would want to use F1 as the help key and so on. Which one of the termination keys that actually interrupted the editing process can be checked by calling GetTermKey.

Input procedures can give a proposal for answers that can be accepted (with a termination key), be ignored (by writing your own characters) or be modified (using the editing keys).

Normally, the keyboard is used for editing. Each window can however be given its own input procedure which returns characters and logical input codes (see the data type InputType). An input procedure can, for example, combine editing via the keyboard with a mouse. See GAEvEd for an example of this.

Additional Functions

By checking the termination key (using GetTermKey) after calling an input procedure, you can incorporate help functions, move between different fields, undo changes etc. See GetTermKey for an example of how this
is used.

User Defined Text Output Procedures

As a default, all text output and input procedures in MGIO use the basic character based output procedure which is found in the unit MGLow (MGLow.DrawChar). However, it is fully possible to install your own, window based character output routines. A new character output procedure for the current windows is set by calling SetDrawCharProc. All Wr and Rd procedures in MGIO will from now on use the new routine.

User Defined Input Procedures

User defined input routines, that can handle other data types as well (e.g. dates, national insurance numbers, etc.), can be constructed by direct calls to the basic string editing routine, StrEdit.

The input parameters are

- the maximum number of characters to input,
- a string variable whose contents, if the string is not empty, is written as a proposal,
- the cursor's start position in the string,
- a set of editing modes (EditModeSet type),
- an input procedure (KbdInput or user defined) of the type InputProc, and
- a string conversion procedure (or NullStrConversion).

All editing functions can be used. Characters and control codes are obtained from the input procedure and each character is checked and converted by the string conversion procedure if this is specified.

Different editing modes (e.g. "erase proposal string", "overwrite text", "insert text", etc.) may be combined.

Editing is interrupted by pressing a key or combination of keys from the set of termination keys in the current window.

The output parameters are the edited string, the cursor's position at termination, and the termination key used.
See **StrEdit** for examples.

**String Editors**

In event based applications serving keyboard as well as mouse and clock events, it is difficult to use I/O routines for input that take complete control until the input is finished. **RdKbd, RdStr, RdReal, StrEdit**, etc. are all examples on such blocking procedures. It is true that you can change the current input procedure to also survey the mouse and clock (see the **GAEvEd** example), but this approach is often perceived as unnatural and may have a detrimental effect on the program design.

In event based programming it is better to use a different approach; first you create a window based string editor, as a variable of the **StrEditor** type, by invoking **CreateStrEditor**. Then you call this editor by invoking **CallStrEditor** from the event loop when relevant events, mostly keyboard or mouse events, occur. **CallStrEditor** updates the string worked on and the screen display. When the editing is finished it is time to call **DisposeStrEditor**.

This technique makes it possible to manage string editing as a natural part of a regular event loop, without blocking other events. You can even keep multiple string editors active at the same time, each editor tied to it's own **StringEditor** variable and in it's own window. See **GADualEd** for an example.

**Examples**

**GA14** illustrates string editing using different character attributes. **GAINput** gives more examples of input of strings and numbers using standard procedures as well as user defined input procedures. **GASTrEd** shows the effect of using different drawing modes and font types. **GAEvEd** is an example of combining keyboard and mouse editing support utilizing a user written input procedure. **GADualEd** illustrates event based programming with two editors running concurrently.
Constants

unitName 'MGIO'
copyright 'DATABITEN'

invalidCursor 1200 Error codes MGIO
maxLengthError 1201
invalidInteger 1202
invalidScrollLines 1203
invalidTermKeySet 1204
invalidInputProc 1205
invalidWord 1206
invalidLongInt 1207
invalidReal 1208
invalidDrawCharProc 1209
invalidAttributes 1215
invalidStrEditor 1220

Specific error codes for MGIO.

defines the following keys and combinations of keys in the actual computer. Under DOS, keys defines the following keys and combinations of keys:

<table>
<thead>
<tr>
<th>keys.field</th>
<th>PC keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs</td>
<td>Backspace</td>
</tr>
<tr>
<td>cr</td>
<td>Enter</td>
</tr>
<tr>
<td>tab</td>
<td>Tab</td>
</tr>
<tr>
<td>If</td>
<td>Ctrl-J</td>
</tr>
<tr>
<td>eof</td>
<td>Ctrl-Z</td>
</tr>
<tr>
<td>esc1</td>
<td>Esc</td>
</tr>
<tr>
<td>esc2</td>
<td>Ctrl-U</td>
</tr>
<tr>
<td>left1</td>
<td>Left arrow</td>
</tr>
<tr>
<td>left2</td>
<td>Ctrl-S</td>
</tr>
<tr>
<td>right1</td>
<td>Right arrow</td>
</tr>
<tr>
<td>right2</td>
<td>Ctrl-D</td>
</tr>
<tr>
<td>ctrlLeft</td>
<td>Ctrl-left arrow</td>
</tr>
<tr>
<td>ctrlRight</td>
<td>Ctrl-right arrow</td>
</tr>
</tbody>
</table>

MultiGraphics & Borland Pascal/Turbo Pascal   DATABITEN
home
endkey
ctrlHome
ctrlEnd
up1
up2
down1
down2
pgUp1
pgUp2
ctrlPgUp
pgDown1
pgDown2
ctrlPgDown
ins1
ins2
del1
del2
ctrlIns
ctrlDel
f1, f2, f3, f4, f5, f6, f7, f8, f9, f10
ctrlf1, ctrlf2, ctrlf3, ctrlf4, ctrlf5,
crlf6, crlf7, crlf8, crlf9, crlf10
shf1, shf2, shf3, shf4, shf5,
shf6, shf7, shf8, shf9, shf10
altf1, altf2, altf3, altf4, altf5,
altf6, altf7, altf8, altf9, altf10

NullDrawChar = DrawCharProc(NIL);

This is the "NULL" procedure to indicate the lack of a character output procedure.

NullInput = InputProc(NIL);

This is the "NULL" procedure to indicate the lack of an input procedure.

NullStrConversion = StrConversionProc(NIL);

This is the "NULL" procedure to indicate the lack of a string conversion procedure. It can be used as a parameter to StrEdit and CreateStrEditor if character conversion is not used.
Data Types

CharAttribute  =
   (underline,bold,italic,shadow,engrave,emboss);

CharAttributeSet = SET OF CharAttribute;

Enumerated type of character attributes and associated set type.

CharAttributePos = RECORD
   dxShadow,
   dyShadow,
   dxEngrave,
   dyEngrave,
   dxEmboss,
   dyEmboss    : ShortInt;
END;
record type to specify the position of certain character attributes.

TermKeySet = SET OF 0..255;
This is a system dependent set of codes for termination keys. The codes are interpreted as follows:

0..31 character values for control keys (Ctrl-C, Enter, etc.)
32..255 Scan code for special keys (numerical, function keys, etc.)

KbdKeys = RECORD..END;
See the keys constant above.

EditMode     =
   (deleteProposal,charInsert,charOverwrite);
Enumeration type specifying the editing modes of the input procedures (StrEdit, CallStrEditor and "Rd" procedures). deleteProposal (default mode) means that the proposal string is erased if the first key pressed following the call of the input procedure is a writeable character. The proposal string is retained if the first key pressed is an editing key. charInsert (also default) allows new characters to be inserted in front of previous characters, while charOverWrite selects overwrite mode.

EditModeSet  = SET OF EditMode;
Set type that makes it possible to specify combinations of different editing modes, for instance `deleteProposal` with `charOverwrite`.

\[
\text{DrawCharProc} = \text{PROCEDURE}( \text{win} : \text{Window}; \text{ch} : \text{CHAR}; \text{VAR error: INTEGER}: (*O*)));
\]

Procedure variable type for character output procedures. See `IOMode` and `SetDrawCharProc`.

\[
\text{InputType} = (\text{leftInput*}, \text{rightInput*}, \text{upInput}, \text{downInput}, \text{pgUpInput}, \text{pgDnInput}, \text{tabInput}, \text{homeInput*}, \text{endInput*}, \text{movetoInput*}, \text{xyInput*}, \text{bsInput*}, \text{delInput*}, \text{insInput}, \text{helpInput}, \text{termInput*}, \text{charInput*}, \text{leftButtonInput}, \text{middleButtonInput}, \text{rightButtonInput}, \text{otherInput})
\]

Defines *logical operations* for editing, and releases text editing routines in `MGIO` from any specific hardware implementation (keyboard, mouse, choice of keys etc.). The values from `InputType` are returned from user defined input procedures of the type `InputProc`. Values marked with asterisks are those interpreted by `StrEdit` och `CallStrEditor`.

\[
\text{InputData} = \text{RECORD} \begin{align*}
\text{editLeft}, \\
\text{editBottom}, \\
\text{editRight}, \\
\text{editTop} & \ : \ \text{INTEGER}; \\
\text{editModes} & \ : \ \text{EditModeSet}; \\
\text{termSet} & \ : \ \text{TermKeySet}; 
\end{align*} \text{END;}
\]

This is data that can be read (and eventually modified) by input procedures - for more information see below. `editLeft..editTop` define the current
extension of the editing area measured in window coordinates. editMode contains the current editing modes and termSet contains the current set of acceptable termination keys.

```pascal
InputProc =
  PROCEDURE (VAR input : InputType; (*O*)
              VAR key : KbdCode;   (*O*)
              VAR inData : InputData; (*IO*)
              VAR editColOrX: INTEGER;   (*IO*)
              VAR editRowOrY: INTEGER    (*IO*));
```

Procedure variable type for input procedures. Input parameters are inData, the current column in editColOrX and the row in editRowOrY (this allows multi-line editing) for the cursor. Output parameters are the current logical input operation in input, the key code in key, modified fields in inData and the new current column and line for the cursor in editColOrX and editRowOrY, or the new current window position of the cursor if input = xyInput.

A standard procedure, KbdInput of the type InputProc is defined in MGIO.

```pascal
StrConversionProc =
  PROCEDURE( newCh : CHAR;
             curStr : String;
             pos    : Word;
             VAR convStr: String (*O*));
```

Procedure variable type for string conversion procedures. For each character input, StrEdit and CallStrEditor calls a conversion procedure to check if the character is to be accepted or not and if it is to be modified. The input parameters are the current character in newCh, the current string in curStr and the position, pos (the first position is 1), in this string where the character is to be inserted. The output parameter is the converted string. This converted string can be empty (""), which means that the character has not been accepted, or contain one or more characters. That is to say one input character can be expanded to several characters.

The latter possibility can be used to "fill in" information, such as when the user presses the key Y in answer to a Yes/No question and the computer writes YES. One other example could be when the computer fills in the punctuation marks in a date field or national insurance number. The computer can also use string conversion procedures to verify if a string is reasonable or valid, e.g., if a date contains correct information or if
a national insurance number is correct.

StrEditor

Data type for string editors. Used by \texttt{CreateStrEditor}, etc.

\texttt{IOMode = RECORD}
\begin{itemize}
\item \texttt{scroll} : BOOLEAN;
\item \texttt{prevCR} : BOOLEAN;
\item \texttt{chAttributes} : CharAttributeSet;
\item \texttt{cursorVisi} : Word;
\item \texttt{cursor} : CHAR;
\item \texttt{cursorColor} : MG.DeviceColor;
\item \texttt{blink} : BOOLEAN;
\item \texttt{beep} : BOOLEAN;
\item \texttt{proposal} : BOOLEAN;
\item \texttt{editModes} : EditModeSet;
\item \texttt{termSet} : TermKeySet;
\item \texttt{termKey} : KbdCode;
\item \texttt{CurDrawChar} : DrawCharProc;
\item \texttt{CurInput} : InputProc;
\item \texttt{dark, light} : MG.RGB;
\item \texttt{attrPos} : CharAttributePos;
\end{itemize}
\texttt{END;}

Data type for \textit{i/o mode} for those windows that have the text \textit{i/o} capability. The current window mode (\texttt{MG.WindowMode}) in these windows (see "MG - Window") is supplemented with the information in \texttt{IOMode}.

\section*{Variables}

\texttt{EP: ErrProc;}

This is the local error procedure for \texttt{MGIO}.

\texttt{revision: Word;}

The current revision for the implementation of the unit \texttt{MGIO}.

\texttt{cap: Capability;
Capability variable for text i/o. It is used as a parameter for MG.EnhanceWin and MG.EnhanceScr to give windows and screens the text i/o capability.

Procedures & Functions

Creating Text Windows

**CreateWin**

```pascal
P CreateWin(VAR win : Window; (*O*)
    x0,y0       : INTEGER;
    width,height: INTEGER);
```

This procedure creates a window and enhances that window with the text i/o capability. \((x0,y0)\) specifies the position for the window's lower left corner, expressed in the screen window's device coordinates. **width** and **height** specify the window's width and height expressed in screen units.

MGIO.CreateWin combines the effect of **MG.CreateWin** and **MG.EnhanceWin**.

The current position is set to column 1, row 1 i.e. near the window's top left corner, expressed in the current font's average width and height.

A window with text i/o capability is removed directly, just like other windows, by calling **MG.DisposeWin**, or indirectly by calling **MG.DisposeScr** or **MG.CloseDevice**.

See **MG.CreateWin** and **MG.EnhanceWin** for more details (e.g. error codes).
VAR
    win: MG.Window;
    ..
    MGIO.CreateWin(win,100,100,200,150);
    IF SV.Result <> SV.ok THEN ..

Window Data

**CurWinCols**

F CurWinCols: INTEGER;

This function returns the number of text columns in the current window, expressed as the average width of the window’s current font modified by the current character attributes.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability.

If an error occurs, MAXINT is returned.

**CurWinRows**

F CurWinRows: INTEGER;

This function returns the number of text rows in the current window, expressed in the height of the window’s current font modified by the current character attributes.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability.

If an error occurs, MAXINT is returned.

MGIO.SetColRow(1, MGIO.CurWinRows);
Position

AdjustCharPos

\texttt{P AdjustCharPos(ch : CHAR;}
\texttt{   hor, vert: INTEGER);};

This procedure adjusts the current position horizontally and vertically \textit{relative} to the present position in the current window.

The displacement in the x direction is carried out using \texttt{hor in units of parts per thousand} of the character width for the character \texttt{ch} modified by the current character attributes. If \texttt{hor} > 0, the current position is moved to the right and if \texttt{hor} < 0, it is moved to the left.

The displacement is carried out in the y direction using \texttt{vert in units of parts per thousand} of the current font height modified by the current character attributes. If \texttt{vert} > 0, the current position is moved upwards and if \texttt{vert} < 0, it is moved downward.

Among other things, the procedure can be used to write exponents and indexes or shadows in graphic text.

SV.Result

\texttt{processWithoutWin} the calling process has no window
\texttt{capDontExist} current window has no text i/o capability

MGIO.AdjustCharPos(ch,-1000,0);
MGIO.WrChar('H');
MGIO.AdjustCharPos(ch,0,-500);MGIO.WrChar('2');
MGIO.AdjustCharPos(ch,0,500); MGIO.WrChar('O');

SetColRow

\texttt{P SetColRow(col, row: INTEGER);};

This procedure sets the current text column and text row in the current window, expressed in the current font's average width and height modified by the current character attributes. The current position in MG is updated.

Columns and rows are numbered from 1 upwards.
When a window is given text capability, the current position is assigned to column 1 and row 1, expressed in the current font. The start position lies near the top left corner of the window. This is in contrast to MG which sets the starting point as the window's lower left corner.

Text is written with the font base line at the height of the current position.

}\sv_result\processWithoutWin\capDontExist
\mgio\setcolrow(1,1);

\textbf{CurCol}

\texttt{F CurCol: INTEGER;}

This function returns the current text column in the current window, expressed in the current font's average width modified by the current character attributes.

Columns are numbered from 1 upwards.

}\sv_result\processWithoutWin\capDontExist
\mgio.curcol > 40 \text{ THEN}

\textbf{CurRow}

\texttt{F CurRow: INTEGER;}

This function returns the current text row in the current window, expressed in the height of the current font modified by the current character attributes.

Rows are numbered from 1 upwards.
SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

If an error occurs, \texttt{MAXINT} is returned.

\begin{tabular}{|p{1.5in}|p{1in}|}
\hline
\texttt{Tab} & DL \\
\hline
\end{tabular}

\texttt{P \ Tab(col: INTEGER);} \\
This procedure moves the current position to the column \texttt{col} on the same row in the window. If the current column > \texttt{col}, nothing is done. Columns are expressed using the current font's average width modified by the current character attributes.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

\texttt{FOR i:= 1 TO 10 DO BEGIN}
\texttt{ MGIO.Tab(10); MGIO.WrInt(i,2);}
\texttt{ MGIO.Tab(20); MGIO.WrFixReal(r[i],3,7);}
\texttt{ MGIO.WrLn;}
\texttt{END;}

\textbf{Widths and Heights of Characters}

\begin{tabular}{|p{1.5in}|p{1in}|}
\hline
\texttt{CurAttrCharWidth} & \\
\hline
\end{tabular}

\texttt{P CurAttrCharWidth(ch: CHAR): INTEGER;}
Returns the width for the character \texttt{ch} in the font of the current window, taken into consideration the current character attributes.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

If an error occurs, \texttt{MAXINT} is returned.
**CurAttrCharBaseline**

```pascal
P CurAttrCharBaseline(): INTEGER;
```

Returns the height for the base line in the font of the current window, taken into consideration the current character attributes.

**SV.Result**
- `processWithoutWin`: the calling process has no window
- `capDontExist`: current window has no text i/o capability

If an error occurs, `MAXINT` is returned.

**CurAttrCharHeight**

```pascal
P CurAttrCharHeight(): INTEGER;
```

Returns the height for the characters in the font of the current window, taken into consideration the current character attributes.

**SV.Result**
- `processWithoutWin`: the calling process has no window
- `capDontExist`: current window has no text i/o capability

If an error occurs, `MAXINT` is returned.

**AttrStrWidth**

```pascal
P AttrStrWidth(s: String): INTEGER;
```

Returns the width in pixels of the string `s` using the current font in the current window, taken into consideration the current character attributes.
If an error occurs, MAXINT is returned.

Clear Window and Line

```
ClrIOWin

P ClrIOWin;
This procedure fills the current window using the current background color in replace mode and sets the current position to column 1, row 1 expressed in the current font's average width and height modified by the current character attributes.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

MG.CreateWin(win,50,50,100,100);
MG.EnhanceWin(win,MGIO.cap);
MG.SetWin(win);
MG.MatchBackground(MG.blue);
MGIO.ClrIOWin;
```

```
ClrEol

P ClrEol;
This procedure fills the current line to the right, from the current column to the right hand side of the viewport, using the current background color in replace mode.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
```
MGIO.SetProposal(FALSE);
MGIO.SetColRow(1,5);
MGIO.WrStr('Give a starting value: ');
column:= MGIO.CurCol;
REPEAT
  MGIO.SetColRow(column,5);
  MGIO.ClrEol;
  MGIO.RdReal(r,0);
UNTIL SV.Result = SV.ok;

The Current Text Mode

GetIOMode

P GetIOMode(VAR mode: IOMode (*O*)));

This procedure returns the current i/o mode for the current window in the variable parameter mode.

The data type IOMode is defined under "Data Types" above.

The procedure is normally used with SetIOMode to save the current i/o mode and then restore it.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

VAR
  oldIOMode: MGIO.IOMode;
  ..
  MGIO.GetIOMode(oldIOMode);
  MGIO.SetCursor('>');
  MGIO.SetCursorBlink(FALSE);
  MGIO.SetBeep(FALSE);
  ..
  MGIO.SetIOMode(oldIOMode);
**SetIOMode**

```pascal
PROCEDURE SetIOMode(VAR mode: IOMode (*I*));
```

This procedure sets the i/o mode in the current window. The window must have the text i/o capability.

*SetIOMode* does not perform checks on data in *mode*.

The procedure is normally used with *GetIOMode* to store and restore a specific i/o mode.

SV.Result
- `processWithoutWin`: the calling process has no window
- `capDontExist`: current window has no text i/o capability

**Scrolling**

**SetScrollMode**

```pascal
PROCEDURE SetScrollMode(onoff: BOOLEAN);
```

This procedure sets the scrolling mode to *on* in the current window if *onoff* is TRUE. It switches it *off* if *onoff* is FALSE.

Scrolling is used by all *Wr* procedures if scrolling is *on*. When a text window is given text i/o capability, the scrolling mode is set to *on*.

SV.Result
- `processWithoutWin`: the calling process has no window
- `capDontExist`: current window has no text i/o capability

**CurScrollMode**

```pascal
FUNCTION CurScrollMode: BOOLEAN;
```

This function returns TRUE if the scrolling mode is *on* in the current window. Otherwise FALSE is returned.

When a text window is given text capability, the scrolling mode is set to *on*. 
SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability.

If an error occurs, **FALSE** is returned.

**Character Attributes**

---

**SetCharAttributes**

\[ \text{P SetCharAttributes}(\text{chAttr: CharAttributeSet}); \]

This procedure is used to establish the current set of character attributes \textit{chAttr} in the current window. These attributes are used by all \textit{Wr} and \textit{Rd} procedures in \textit{MGIO}, but only when the drawing mode \textit{replace} or \textit{stamp} is active. Different attributes can be combined, with the exception of \textit{shadow}, \textit{engrave} and \textit{emboss}.

When a text I/O window is created the character attribute set is empty.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability.
invalidAttributes invalid attribute or combination of attributes

MGIO.SetCharAttributes([MGIO.bold,
MGIO.emboss]);
WriteLn('In the pre-graphics days it was hard to be');
WriteLn('an engraver');

---

**GetCharAttributes**

\[ \text{P GetCharAttributes(VAR chAttr: CharAttributeSet (*O*))}; \]

Reads and returns in \textit{chAttr} the current character attributes of the current window.

SV.Result
processWithoutWin the calling process has no window
capDontExist

current window has no text i/o capability.

\[
\text{MGIO}.\text{GetCharAttributes}(\text{chAttr})
\]
\[
\text{chAttr} := \text{chAttr} + \lfloor \text{MGIO.underline} \rfloor
\]
\[
\text{MGIO}.\text{SetCharAttributes}(\text{chAttr})
\]

**SetCharAttributePos**

P \text{SetCharAttributePos}(\text{pos: CharAttributePos});

This procedure changes the position of the character attributes \text{MGIO\_shadow}, \text{engrave} and \text{emboss} for text in the current window. The new position is specified by the record \text{pos}, specifying the displacements, measured in pixels, of the attributes relative to the characters.

When a text I/O window is created, as a default, the attributes are displaced by 1/500 of the current screen width for \text{shadow} and with 1/1000 of the current screen width for \text{engrave} and \text{emboss} - at least 1 pixel.

SV.Result

\text{processWithoutWin}

the calling process has no window

\text{capDontExist}

current window has no text i/o capability.

\[
\text{VAR}
\]
\[
\begin{array}{c}
\text{win} \quad : \text{MG.Window}; \\
\text{oldAttPos,} \\
\text{attrPos} \quad : \text{MGIO.CharAttributePos}; \\
\end{array}
\]

\[
\begin{array}{c}
\ldots \\
\text{MG.SetWin}(\text{win}); \\
\text{MGIO.GetCharAttributePos}(\text{oldAttPos}); \\
\text{WITH attrPos DO BEGIN} \\
\quad \text{dxShadow} := -5; \\
\quad \text{dyShadow} := -5; \\
\quad \text{dxEngrave} := -2; \\
\quad \text{dyEngrave} := -2; \\
\quad \text{dxEmboss} := -2; \\
\quad \text{dyEmboss} := -2; \\
\text{END; (*with*)} \\
\text{MGIO.SetCharAttributePos}(\text{attrPos}); \\
\text{MGIO.SetCharAttributes(\lfloor \text{MGIO.shadow} \rfloor );} \\
\text{MGIO.WrStr('Shadow to the left and below');} \\
\ldots \\
\text{MGIO.SetCharAttributePos(\text{oldAttPos});}
\end{array}
\]
GetCharAttributePos

P GetCharAttributePos(VAR pos: CharAttributePos (*O*));

Reads and returns in the record pos the current character attribute positions for the current window.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability.

SetCharAttributeColors

P SetCharAttributeColors(dark, light: RGB);

Sets the current attribute colors dark and light of the current window. The colors are specified as RGB values. dark is used for shadows when writing characters with shadow or emboss attributes, while light is used for "illuminated" parts of characters with the attributes engrave or emboss.

When a text I/O window is created the dark color is set to MG.darkGray and light to MG.white.

Internally, dark and light are translated to device dependent colors by help of MG.MatchingForeground. Therefore, the mapping on monochrome devices is dependent on the current "black-and-white" mode.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability.
invalidRGB invalid RGB values

MGIO.SetCharAttributeColors(MG.blue,
MG.lightCyan);

GetCharAttributeColors

P GetCharAttributeColors(VAR dark, light: RGB (*O*));

Reads and returns the current character attribute colors of the current window.


**SetUnderline**

P SetUnderline(onoff: BOOLEAN);

This procedure sets underlining mode *on* in the current window if `onoff` is TRUE. It switches off underlining if `onoff` is FALSE.

The underlining mode is used by all Wr procedures if scrolling is on. When a text window is given text capability, the underlining mode is set to off.

This command is maintained for compatibility reasons with older versions of MultiGraphics. The preferred method is to use SetCharAttributes.

**CurUnderline**

F CurUnderline: BOOLEAN;

This function returns TRUE if the underlining mode is on in the current window. Otherwise FALSE is returned.

When a text window is given text capability, the underlining mode is set to off.

This command is maintained for compatibility reasons with older versions of MultiGraphics. The preferred method is to use GetCharAttributes.

If an error occurs, FALSE is returned.
The Cursor

SetCursor

\textbf{P} \texttt{SetCursor(cursor: CHAR);} \\
This procedure sets the current character for the text (input) cursor in the current window. The cursor is shown when carrying out input procedures that are based on \texttt{StrEdit} (and if \texttt{ShowCursor} has been called). The cursor character is drawn using the current font.

When a text window is given text capability, the current cursor character is set to \_' (ASCII 95).

\textbf{SV.Result} \\
\texttt{processWithoutWin} the calling process has no window \\
\texttt{capDontExist} current window has no text i/o capability \\
\texttt{invalidCursor} the character is not in the current font

\texttt{MGIO.SetCursor('+');}

CurCursor

\textbf{F} \texttt{CurCursor: CHAR;} \\
This function returns the current character for the cursor in the current window.
If an error occurs, the character **NULL** (ASCII 0) is returned.

### ShowCursor

```
P ShowCursor;
```

This procedure increases the visibility of the text cursor by one step. However, the cursor only appears on the screen when it has reached the level of "full visibility".

When a text window is given text capability, the text cursor is fully visible.

### HideCursor

```
P HideCursor;
```

This procedure decreases the visibility of the text cursor by one step. The cursor only appears on the screen when it has reached the level of "full visibility".

When a text window is given text capability, the text cursor is fully visible.

### SV.Result

- **processWithoutWin**: the calling process has no window
- **capDontExist**: current window has no text i/o capability
- **invalidCursor**: the character is not in the current font
**CurShowCursor**

**F CurShowCursor: BOOLEAN;**

This function returns **TRUE** if the cursor is "fully visible", otherwise **FALSE**.

**SV.Result**
**processWithoutWin** the calling process has no window
**capDontExist** current window has no text i/o capability

If an error occurs, **FALSE** is returned.

**DrawCursor**

**P DrawCursor;**

Draws the current cursor character at the current position using the **complement** drawing mode. The current position is not changed.

This procedure is useful when you are creating your own input procedures of **InputProc** type.

**SV.Result**
**processWithoutWin** the calling process has no window
**capDontExist** current window has no text i/o capability

**SetCursorColor**

**P SetCursorColor(color: DeviceColor);**

This procedure sets the device color for the cursor in the current window.

When a text window is given text capability, the cursor's device color is set to the maximum permitted device color (usually "white") in the current screen.
SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
invalidDeviceColor invalid device color

CurCursorColor

F CurCursorColor: DeviceColor;

This function returns the current device color for the cursor in the current window.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

If an error occurs, maxDeviceColor is returned.

SetCursorBlink

P SetCursorBlink(onoff: BOOLEAN);

This procedure sets cursor blinking to on if the parameter onoff is TRUE. It switches off cursor blinking if the parameter onoff is FALSE.

When a text window is given text capability, cursor blinking is set to on.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

CurCursorBlink

F CurCursorBlink: BOOLEAN;

This function returns TRUE if cursor blinking is on, otherwise FALSE is returned.
Warning Signals (Beeps)

**SetBeep**

```pascal
procedure SetBeep(onoff: BOOLEAN);
```

This procedure sets the warning beep to `on` if the parameter `onoff` is `TRUE`. It is switched `off` if the parameter `onoff` is `FALSE`.

When a text window is given text capability, the warning beep is set to `on`.

**CurBeep**

```pascal
function CurBeep: BOOLEAN;
```

This function returns `TRUE` if the warning beep is `on`, otherwise `FALSE` is returned.
Beep

P Beep;

This procedure produces a warning beep.

Beep in turn calls SV.Beep.

Proposal Mode

SetProposal

P SetProposal(onoff: BOOLEAN);

This procedure sets proposal mode to on for Rd procedures if the parameter onoff is TRUE. It switches proposal mode off if the parameter onoff is FALSE.

Proposal mode is used when calling RdChar, RdStr, RdWord, RdInt, RdLong, RdReal and RdFixReal.

When a text window is given text capability, proposal mode is set to on.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o

MGIO.SetProposal(TRUE);
i:= 1;
MGIO.RdInt(i);

CurProposal

F CurProposal: BOOLEAN;

This function returns TRUE if proposal mode is on, otherwise FALSE is returned.

Proposal mode is used when calling RdChar, RdStr, RdWord, RdInt, RdLong, RdReal and RdFixReal.
Ref 23-34

MGIO - Text I/O

**SV.Result**
- **processWithoutWin** the calling process has no window
- **capDontExist** current window has no text i/o capability

If an error occurs, **FALSE** is returned.

**SetEditModes**

```pascal
P SetEditModes(modes: EditModeSet);
```

Defines a set of editing modes for the **Rd** procedures in the current window.

The editing modes are used when calling **RdChar, RdStr, RdCard, RdInt, RdLong, RdReal** and **RdFixReal**.

When a text window is given text capability, the editing modes are set to:
```
[deleteProposal, charInsert]
```

This default mode makes proposal strings to be erased if the first key pressed is not among the editing keys. It also makes new text to be inserted in front of previous text.

**SV.Result**
- **processWithoutWin** the calling process has no window
- **capDontExist** current window has no text i/o capability

```
MGIO.SetEditModes([deleteProposal, charOverwrite]);
i:= 5;
MGIO.RdInt(i);
```

**GetEditModes**

```pascal
P GetEditModes(VAR modes: EditModeSet (*0*));
```

Returns in **modes** the current editing modes of the current window. The procedure is often used with **SetEditModes** to save and restore editing modes.
SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

Termination Keys

**SetTermKeySet**

\[
P \text{SetTermKeySet}(\text{termSet: TermKeySet});
\]

This procedure installs the set `termSet` with termination key codes in the current window. The key codes specify those keys, or combinations of keys, that terminate input operations in the procedures `StrEdit`, `RdChar`, `RdStr`, `RdWord`, `RdInt`, `RdLong`, `RdReal` and `RdFixReal`.

`TermKeySet` is described under "Data Types" above. The following editing keys *cannot* be used as termination keys:

- left arrow, right arrow, `<Home>`, `<End>` and `<Del>`.

When a text window is given text capability, the set of termination keys in the PC environment is assigned to:

- **Ctrl-C, Enter, Ctrl-Z, Esc**
  .i.e. [3,13,26,27].

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
invalidTermKeySet the set is empty

(* <Enter>, <Esc> & <F1> *)
MGIO.SetTermKeySet([13,27,59]);
(*$V-**)MGIO.RdStr(s,SizeOf(s) - 1);(*$V++*)
MGIO.GetTermKey(key);
IF key.wrd = keys.f1.wrd THEN ..
GetTermKeySet

P GetTermKeySet(VAR termSet: TermKeySet (*O*));

This procedure returns, in the variable parameter `termSet`, the set of termination key codes that are used in the current window.

The keys are used to stop input in the procedures `StrEdit, RdChar, RdStr, RdWord, RdInt, RdLong, RdReal` and `RdFixReal`.

SV.Result

processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

If an error occurs, the empty set is returned.

Input Procedures

SetInputProc

P SetInputProc(UserInput: InputProc);

This procedure sets the current input procedure of the `InputProc` type in the current window. This procedure is called by `RdChar, RdStr, RdWord, RdInt, RdLong, RdReal` and `RdFixReal`.

When a window is given text capability, the input procedure is set to `MGIO.KbdInput` as a default.

SV.Result

processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

invalidInputProc invalid input procedure.

`MGIO.SetInputProc(MouseInput);`
**GetInputProc**

\[ P \text{ GetInputProc}(\text{VAR UserInput: InputProc \(*O*\)}) \];

This procedure returns, in the variable parameter UserInput, the current
input procedure in the current window.

The procedure is normally used with SetInputProc to store the current
input procedure and later reinstall it.

**SV.Result**

processWithoutWin the calling process has no window
capDontExist current window has no text i/o
capability

If an error occurs, MGIO.NullInput is returned.

\[ \text{VAR OldInput: MGIO.InputProc;} \]
\[ \ldots \]
\[ \text{MGIO.GetInputProc(OldInput);} \]
\[ \text{MGIO.SetInputProc(NewInput);} \]
\[ \ldots \]
\[ \text{MGIO.SetInputProc(OldInput);} \]

**SetDrawCharProc**

\[ P \text{ SetDrawCharProc(DrawChar: DrawCharProc);} \];

This procedure sets the current character output procedure of the
DrawCharProc type in the current window. This procedure is called by all
Wr- and Rd-procedures.

When a window is enhanced with text capability, the output procedure is set
to MGLow.DrawChar as a default.

**SV.Result**

processWithoutWin the calling process has no window
capDontExist current window has no text i/o
capability
invalidDrawCharProc invalid character output procedure.

MGIO.SetDrawCharProc(DrawShadow);

**GetDrawCharProc**

P GetDrawCharProc(VAR UserDrawChar: DrawCharProc (*O*));

This procedure returns, in the variable parameter DrawChar, the current character output procedure in the current window.

The procedure is normally used with **SetDrawCharProc** to store the current output procedure and later reinstall it.

SV.Result

processWithoutWin the calling process has no window
capDontExist current window has no text i/o
capability

If an error occurs, MGIO.NullDrawChar is returned.

VAR

  OldDrawChar: MGIO.DrawCharProc;
  ..
  MGIO.GetDrawCharProc(OldDrawChar);
  MGIO.SetDrawCharProc(NewDrawChar);
  ..
  MGIO.SetDrawCharProc(OldDrawChar);

**Text Output**

**WrLn**

P WrLn;

This procedure starts a new text line (row) in the current window. The new current position is the first column of the row below the current row. If the current row is the last row in the window, the contents of the window is scrolled up a row. The last row of the window is filled using the current background color.

The row height is of the same size as the font height of the current font.
SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

**WrChar**

P WrChar(ch: CHAR);

This procedure writes the character ch at the current position in the current window, using the current window mode, i.e. viewport, drawing mode, foreground and background colors, color mask and font.

If the scrolling mode is on (see **SetScrollMode**), a new line is started automatically when printing reaches the window's right hand side, the same applies to scrolling when printing reaches the bottom of the window. The current position is updated with the width of the character. If underlining mode is on, the character is underlined.

If the scrolling mode is off, the text output is clipped against the window's right hand side. The current position is updated however with the width of the character even if the text is clipped.

Text output is faster if scrolling is switched off.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability

Errors are also reported from **MG.DrawStr** and the virtual memory handler in **VM** and **AutoVM**, etc.

**WrStr**

P WrStr(s: String);

This procedure writes the text string s in the current window using the current window mode, i.e. starting at the current position with subsequent characters written in the current direction, using the current font, foreground and background colors, drawing mode, color mask and taking into consideration the current viewport.
If the scrolling mode is *on* (see `SetScrollMode`), new lines (rows) are created automatically when the printing reaches the window's right hand side. Similarly, scrolling is performed when the printing reaches the bottom of the window. The current position is updated with the width of the string. If underline mode is *on*, the string is underlined.

If the scrolling mode is *off*, the text output is clipped against the window's right hand side. The current position is updated however with the width of the string even if the text is clipped.

Text output is faster if scrolling is switched off.

```
SV.Result
processWithoutWin     the calling process has no window
capDontExist         current window has no text i/o
capability
```

Errors are also reported from `MG.DrawStr` and the virtual the memory handler in `VM`, `AutoVM`, etc.

```
MGIO.WrStr('Press <Enter>');
```

<table>
<thead>
<tr>
<th>WrWord</th>
<th>DL</th>
</tr>
</thead>
</table>
| `P WrWord(c : Word;  
  width: Word);` |

This procedure writes the non-negative integer `c` in the current window using the current window mode, i.e. starting at the current position with subsequent characters written in the current direction, using the current font, foreground and background colors, drawing mode, color mask and taking into consideration the current viewport.

*width* specifies the minimum number of character positions that are to be used. If more positions are needed, the width of the field will be increased automatically. If fewer positions are required, the number is right justified and is preceded by blanks.

If the scrolling mode is *on* (see `SetScrollMode`), new lines (rows) are created automatically when the printing reaches the window's right hand side. Similarly, scrolling is performed when the printing reaches the bottom of the window. The current position is updated with the width of the string. If underline mode is *on*, the string is underlined.
If the scrolling mode is *off*, the text output is clipped against the window's right hand side. The current position is updated however with the width of the string even if the text is clipped.

Text output is faster if scrolling is switched off.

Errors are also reported from *WrStr*.

```
MGIO.WrWord(i,10);
```

<table>
<thead>
<tr>
<th>WrInt</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>p WrInt(i : INTEGER; width: Word);</td>
<td></td>
</tr>
</tbody>
</table>

This procedure writes the integer \(i\) in the current window using the current window mode, i.e. starting at the current position with subsequent characters written in the current direction, using the current font, foreground and background colors, drawing mode, color mask and taking into consideration the current viewport.

*width* specifies the minimum number of character positions that are to be used. If more positions are needed, the width of the field will be increased automatically. If fewer positions are required, the number is right justified and is preceded by blanks.

If the scrolling mode is *on* (see *SetScrollMode*), new lines (rows) are created automatically when the printing reaches the window's right hand side. Similarly, scrolling is performed when the printing reaches the bottom of the window. The current position is updated with the width of the string. If underline mode is *on*, the string is underlined.

If the scrolling mode is *off*, the text output is clipped against the window's right hand side. The current position is updated however with the width of the string even if the text is clipped.

Text output is faster if scrolling is switched off.

Errors are also reported from *WrStr*.

```
MGIO.WrInt(i,0);
```
**WrLong**

```pascal
P WrLong(li : LongInt;
           width: Word);
```

This procedure writes the long integer `li` in the current window using the current window mode, i.e. starting at the current position with subsequent characters written in the current direction, using the current font, foreground and background colors, drawing mode, color mask and taking into consideration the current viewport.

`width` specifies the minimum number of character positions that are to be used. If more positions are needed, the width of the field will be increased automatically. If fewer positions are required, the number is right justified and is preceded by blanks.

If the scrolling mode is *on* (see `SetScrollMode`), new lines (rows) are created automatically when the printing reaches the window's right hand side. Similarly, scrolling is performed when the printing reaches the bottom of the window. The current position is updated with the width of the integer string. If underline mode is *on*, the integer is underlined.

If the scrolling mode is *off*, the text output is clipped against the window's right hand side. The current position is updated however with the width of the integer string even if the text is clipped.

Text output is faster if scrolling is switched off.

Errors are also reported from `WrStr`.

```pascal
MGIO.WrLong(long,10);
```

**WrReal**

```pascal
P WrReal(r : Real;
          digits,width: Word);
```

This procedure writes the floating point number `r` in the current window using the current window mode, i.e. starting at the current position with subsequent characters written in the current direction, using the current font, foreground and background colors, drawing mode, color mask and taking into consideration the current viewport. The number is written in *exponent format*. 
digits specifies the number of figures in the mantissa (the number of significant figures).

width specifies the minimum number of character positions that are to be used. If more positions are needed, the width of the field will be increased automatically. If fewer positions are required, the number is right justified and is preceded by blanks.

If the scrolling mode is on (see SetScrollMode), new lines (rows) are created automatically when the printing reaches the window's right hand side. Similarly, scrolling is performed when the printing reaches the bottom of the window. The current position is updated with the width of the floating point string. If underline mode is on, the floating point number is underlined.

If the scrolling mode is off, the text output is clipped against the window's right hand side. The current position is updated however with the width of the floating point string even if the text is clipped.

Errors are also reported from WrStr.

It reports errors from WrStr.

MGIO.WrReal(r, 6, 12);

<table>
<thead>
<tr>
<th>WrFixReal</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>P WrFixReal(r : Real; decimals, width: Word);</td>
<td></td>
</tr>
</tbody>
</table>

This procedure writes the floating point number r in the current window using the current window mode, i.e. starting at the current position with subsequent characters written in the current direction, using the current font, foreground and background colors, drawing mode, color mask and taking into consideration the current viewport. The number is written in decimal format with a fixed number of decimals.

decimals specifies the number of decimals.

width specifies the minimum number of character positions that are to be used. If more positions are needed, the width of the field will be increased automatically. If fewer positions are required, the number is right justified and is preceded by blanks.
If the scrolling mode is on (see SetScrollMode), new lines (rows) are created automatically when the printing reaches the window's right hand side. Similarly, scrolling is performed when the printing reaches the bottom of the window. The current position is updated with the width of the floating point string. If underline mode is on, the string is underlined.

If the scrolling mode is off, the text output is clipped against the window's right hand side. The current position is updated however with the width of the floating point string even if the text is clipped.

Printing is faster if scrolling is switched off.

Errors are also reported from WrStr.

It reports errors from WrStr.

MGIO.WrFixReal(r, 3, 8);

<table>
<thead>
<tr>
<th>Scroll</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>P Scroll(lines: INTEGER);</td>
<td></td>
</tr>
</tbody>
</table>

This procedure takes the contents of the current window and copies ("scrolls") it the number of pixel rows upwards in the y direction specified in the parameter lines and fills the lowermost lines with the current background color.

SV.Result processWithoutWin capDontExist invalidScrollLines the calling process has no window current window has no text i/o capability. only positive values are valid in lines
Text Input

Delays

**Wait**

```pascal
PROCEDURE Wait(ms : Word;
VAR userBreak: BOOLEAN (*O*));
```

This procedure produces a delay of approximately `ms` milliseconds, if the procedure is not interrupted by a key being pressed. In which case, the variable parameter `userBreak` is set to `TRUE`. If no key has been pressed, `userBreak` is set to `FALSE` when the delaying time has elapsed.

SV.Result always returns SV.ok.

```pascal
MGIO.Wait(10000,userBreak);
IF userBreak THEN ..
```

Reading the Keyboard

**RdKbd**

```pascal
PROCEDURE RdKbd(VAR key: KbdCode (*O*));
```

This procedure waits for a keyboard key to be pressed and then reads the key. The key code or the combination of key codes is returned in the variable parameter `key` of the system dependent type `KbdCode`.

If the cursor is *on*, it is displayed at the current position. No characters are echoed to the screen. Input operations do not need to be terminated with `Enter`.

SV.Result

- `processWithoutWin` the calling process has no window
- `capDontExist` current window has no text i/o capability

If an error occurs, the key code `nullKey` is returned.
PROCEDURE PressEnter;
VAR
  key: MG.KbdCode;
BEGIN
  REPEAT
    MGIO.RdKbd(key);
    UNTIL key.wrd = keys.cr.wrd;
END; (*PressEnter*)

Editing Strings

<table>
<thead>
<tr>
<th>StrEdit</th>
</tr>
</thead>
</table>
| P StrEdit(VAR s : String; (*IO*)
  VAR cursorPos: Word; (*IO*)
  maxLen : Word;
  modes : EditModeSet;
  UserInput : InputProc;
  Convert : StrConversionProc;
  VAR termKey : KbdCode (*O*)) |

This is the basic procedure for input and editing of strings in non event based applications. StrEdit is called by all "Rd" procedures in MGIO.

When a call is made, s can be an empty string or contain a proposal for a reply string. When leaving StrEdit s holds the user reply string.

cursorPos is set to the cursor's start position in the proposal string. The first character is counted as character 1! If cursorPos > the length of the string s, the start position is assigned to the first position after the last character in the string.

maxLen is the maximum permitted number of characters in the reply string. The cursor cannot move beyond this limit when editing is in progress. However, if the cursor reaches the window's right hand side before maxLen, the cursor is stopped at the window's boundary.

modes is a set expression of type EditModeSet holding the desired editing modes of EditMode type. This parameter decides if the proposal string is erased or not when the first key pressed is a character key (deleteProposal) and if StrEdit shall use overwrite mode (charOverwrite) or insert mode (charInsert).
**UserInput** is a parameter of procedure type which specifies the desired input routine of the type **InputProc**. When using simple keyboard input, **KbdInput** in MGIO is normally used.

**Convert** is a parameter of procedure type which specifies the desired conversion routine of the type **StrConversionProc**. If the constant **NullStrConversion** in MGIO is used, no character conversion is carried out.

**termKey** is a variable parameter which, after the call, contains the code for the termination key or combination of keys.

Using **StrEdit**, it is possible to tailor user defined input routines for text strings.

**StrEdit** begins, if $s <> ''$, by writing "the proposal string" and placing the cursor in the character position **cursorPos**. The cursor is shown only if it is on. Characters and editing commands are obtained from the procedure parameter **UserInput**. Each character is sent first to the procedure **Convert**, where it may be modified, before it is placed in the string $s$. When overwrite mode is active, new characters will overwrite old text. When insert mode is active new characters are inserted in front of old text. Input operations are terminated by using a termination key. The key code for this is to be found in **termKey** after the call is made. The edited string is held in $s$.

If the drawing mode is not **replace**, the screen background is automatically saved under the edited text string. If the edited string shrinks, the original screen background is restored.

If the scrolling mode is **on** and the editing is terminated using **Enter**, the current position moves to the beginning of the next text line (row).

**SV.Result**
- **processWithoutWin** the calling process has no window
- **capDontExist** current window has no text i/o capability
- **invalidFont** error in the current window font
- **maxLengthError** error in value in **maxLen**
- **invalidInputProc** invalid input procedure.

Errors are also reported from **MG.CreateVirtScr**, **MG.DisposeScr**, **MG.CopyRect** and **MG.DrawStr**.
PROCEDURE YesOrNoCheck(ch : CHAR;
                      curStr : String;
                      position: Word;
                      VAR convStr : String (*O*));

BEGIN
  IF (position = 1) AND
    ((UpCase(ch) = 'Y') OR (UpCase(ch) = 'N')) THEN
    convStr:= UpCase(ch)
  ELSE
    convStr:= '';
  END; (*YesOrNoCheck*)

FUNCTION Yes(prompt: String): BOOLEAN;
VAR
  chStr        : STRING[1];
  cursorPos    : Word;
  termKey      : MGIO.KbdCode;
  oldScrollMode: BOOLEAN;
  UserInput    : MGIO.InputProc;
BEGIN
  MGIO.WrStr(prompt);
  oldScrollMode:= MGIO.CurScrollMode;
  MGIO.SetScrollMode(FALSE); (*no line feed in StrEdit*)
  MGIO.GetInputProc(UserInput);
  cursorPos:= 1;
  chStr:='';
  REPEAT
    MGIO.StrEdit(chStr,
                  cursorPos,
                  1,
                  [MGIO.charOverwrite],
                  UserInput,
                  YesOrNoCheck,
                  termKey);
    UNTIL (chStr[1] = 'Y') OR (chStr[1] = 'N');
  IF chStr[1] = 'Y' THEN
    MGIO.WrStr('Yes')
  ELSE
    MGIO.WrStr('No');
  END; (*Yes*)
..  
  IF Yes('Yes or No: ') THEN ..

This example shows how you can write your own input reading procedure, in this cases a function that only accepts the answers 'Y' or 'N'.

MultiGraphics & Borland Pascal/Turbo Pascal

DATABITEN
CreateStrEditor

P CreateStrEditor(VAR strEd : StrEditor; (*O*))
VAR s : String; (*IO*)
cursorPos : CARDINAL;
maxLen : CARDINAL;
Convert : StrConversionProc;
VAR edData : InputData (*O*));

Creates a new string editor linked to strEd in the current window. strEd is then used as a parameter to CallStrEditor. The editor is created in the current window and each window can only run one editor at the same time.

When a call is made, s can be an empty string or contain a proposal for a reply string. s is then used as a parameter to CallStrEditor.

cursorPos is set to the cursor's start position in the proposal string. The first character is counted as character 1! If cursorPos > the length of the string s, the start position is assigned to the first position after the last character in the string.

maxLen is the maximum permitted number of characters in the reply string. The cursor cannot move beyond this limit when editing is in progress. However, if the cursor reaches the window's right hand side before maxLen, the cursor is stopped at the window's boundary.

modes is a set expression of typeEditModeSet holding the desired editing modes ofEditMode type. This parameter decides if the proposal string is erased or not when the first key pressed is a character key (deleteProposal) and if StrEdit shall use overwrite mode (charOverwrite) or insert mode (charInsert).

Convert is a parameter of procedure type which specifies the desired conversion routine of the type StrConversionProc. If the constant NullStrConversion in MGIO is used, no character conversion is carried out.

edData is a record of type InputData, which after the call contains information about the current editing region and set of terminating keys. The information can be used to interpret mouse events and keyboards events.
SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
invalidFont error in the current window font
maxLengthError error in value in maxLen

Errors are also reported from MG.CreateVirtScr, MG.DisposeScr, MG.CopyRect and MG.DrawStr.

**CallStrEditor**

```pascal
P CallStrEditor(    strEd        : StrEditor;(*Io*)
                  VAR s            : String; (*IO*)
                  action       : InputType;
                  key          : KbdCode;
                  editStrPosOrX: INTEGER;
                  modes        : EditModeSet;
                  VAR edData       : InputData (*O*));
```

Activates the string editor **strEd** for revising of the string **s** with a new input event described by the parameters **action** for a logical input type, **key** for a keyboard code, **editStrPosOrX** for a current editing position and the set **modes** for editing modes. The editor can be called from other windows than the editor window.

**s** should contain a response string from a preceding call of **CallStrEditor** or **CreateStrEditor**.

*Do not modify s between calls of CallStrEditor!*

**action** contains a logical input code from the enumerated type **InputType**. If the event is of keyboard type you can call **KeyAction** to get the logical input code.

If the action type is **charInput** or **termInput**, **key** shall specify a keyboard code.

**editStrPosOrX** is used to specify a new editing position if **action** has the value **moveToPos** or **xyInput**. The editing position is specified as a new string position (**moveToPos**) or an x windows coordinate for the new editing position (**xyInput**).
modes is a set expression of the type EditModeSet expressing the desired editing modes of EditMode type. At the first call of CallStrEditor, you can use modes to decide if a proposal string shall be erased, in the case the first input code is a character (deleteProposal), or not. During subsequent calls, modes can specify overwrite mode (charOverwrite) or insert mode (charInsert).

edData is a record of type InputData, which after the call contains information about the current editing region and set of termination keys. The information can be used to interpret mouse events and keyboards events.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o
capability
invalidStrEditor invalid strEd parameter

Errors are also reported from MG.CopyRect and MG.DrawStr.

Here is an example of a simple keyboard feeded editor. You will find a more complete example in GADualEd.

VAR
editor : MGIO.StrEditor;
s : ARRAY OF STRING[40];
edData : MGIO.InputData;
edModes : MGIOEditModeSet;
edIdx0 : Word;
key : MG.KbdCode;
action : MG.IO.InputType;
edStrPosOrX : INTEGER;
..edModes:= [];
s:= '';MGIO.CreateStrEditor(editor,
s,MAXINT,SizeOf(s) - 1,
MGIO.NullStrConversion,
edData);
REPEAT
MGIO.RdKbd(key);
action:= MG.IO.KeyAction(key,edData.termSet);
IF action = MG.IO.insInput THEN BEGIN
IF MG.IO.charOverwrite IN edModes THEN BEGIN
edModes:= edModes - [MG.IO.charOverwrite];
MG.IO.SetCursor('_');
END
END
ELSE BEGIN
   edModes:= edModes + [MGIO.charOverwrite];
   MGIO.SetCursor('Û');
END;
END;
ELSE
   MGIO.CallStrEditor(editor, s, action, key, editStrPosOrX, edModes, edData);
UNTIL action = MGIO.termInput;
MGIO.DisposeStrEditor(editor);

DisposeStrEditor

P DisposeStrEditor(VAR strEd: StrEditor (*IO*));

Deinstalls the font editor strEd. Always call this procedure after a terminated string editing session using CallStrEditor.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
invalidStrEditor invalid strEd parameter

Errors are also reported from MG.DisposeScr.

Keyboard Input Procedure

KbdInput

P KbdInput(VAR input : InputType; (*O*)
   VAR key : KbdCode;   (*O*)
   VAR inData : InputData; (*IO*)
   VAR editColOrX: INTEGER;   (*IO*)
   VAR editRowOrY: INTEGER    (*IO*));

This is the default procedure, of the type InputProc, for keyboard oriented editing.
When a window is given the text i/o capability, \texttt{KbdInput} is installed automatically as the current input routine. This is called by \texttt{RdChar}, \texttt{RdStr}, \texttt{RdWord}, \texttt{RdInt}, \texttt{RdLong}, \texttt{RdReal} and \texttt{RdFixReal}. The current input procedure can be changed using \texttt{SetInputProc}.

\texttt{KbdInput} can also be used as a parameter when making a direct call to \texttt{StrEdit}.

\textit{KbdInput is not intended to be called directly by the user, but only via "Rd" procedures or StrEdit.}

This procedure does not set the result code in \texttt{SuperVisor}.

\begin{verbatim}
MGIO.StrEdit(s, pos, 80, [MGIO.charInsert], MGIO.KbdInput, NullStrConversion, termKey);
\end{verbatim}

### KeyAction

\begin{verbatim}
P KeyAction(    key    : KbdCode;
        VAR termSet: TermKeySet (*I*)): InputType;
\end{verbatim}

Default translation of keyboard codes in \texttt{key} to logical input codes. To assist the translator the function also needs the set of current termination keys in \texttt{termSet} (you can get them from \texttt{GetTermKeySet}). \texttt{KeyAction} is called from \texttt{KbdInput}.

The primary usage of \texttt{KeyAction} is to support event based string editors of type \texttt{StrEditor} to translate keyboard events to logical input codes. See \texttt{GADualEd} for an example.

This function does not set the result code in \texttt{SuperVisor}.
Control Procedures for Standard Types

**WordCheck**

```pascal
P WordCheck( ch : CHAR;
curStr : String;
position: Word;
VAR convStr : String (*O*));
```

This is a procedure of the type `StrConversionProc` for editing non-negative integers. It is used by the procedure `RdWord` and can be sent as a parameter to `StrEdit`.

The permitted characters are '0'..'9' and leading blanks.

**IntCheck**

```pascal
P IntCheck( ch : CHAR;
curStr : String;
position: Word;
VAR convStr : String (*O*));
```

This is a procedure of the type `StrConversionProc` for editing of integers. It is used by the procedures `RdInt` and `RdLong`, and can also be sent as a parameter to `StrEdit`.

The permitted characters are '+', '-', '0'..'9' as well as leading blanks. Plus and minus signs can only be written in the first position of the editing string.

```pascal
MGIO.StrEdit(intStr,
pos,
6,
[MGIO.charInsert»],
KbdInput,
MGIO.IntCheck,
termKey);
```
**RealCheck**

```pascal
P RealCheck(ch : CHAR;
curStr : String
position: Word;
VAR convStr : String (*O*));
```

This is a procedure of the type **StrConversionProc** for the editing of floating-point numbers. It is used by the procedures **RdReal** and **RdFixReal**, and can also be sent as a parameter to **StrEdit**.

The permitted characters are '+', '-', ',', '0'..'9', 'E', 'e' and leading blanks. Plus and minus signs can only be written in the first position of the editing string and immediately after 'E' or 'e'.

**Reading Procedures for Standard Types**

**RdChar**

```pascal
P RdChar(VAR ch: CHAR (*IO*));
```

This procedure edits and returns a character in `ch`. Characters are echoed to the screen starting at the current position. Input operations are terminated using any of the current termination keys in the current window.

If the proposal mode is on, the character in `ch` is written before editing. The cursor is shown if the cursor is on.

Characters are written using the current window mode as the drawing mode, foreground and background colors, color mask etc. For input of characters and control codes, the current input procedure is called (as a default **KbdInput**).

If the scrolling mode is on and the editing is terminated using **Enter**, the current position moves to the beginning of the next text line (row).

The key code for the termination key or combination of keys used can be read with **GetTermKey**.
SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o
capability

Errors from StrEdit are also reported.

**RdStr**

```pascal
P RdStr(VAR s : String;(*IO*)
    maxlen: Word);
```

This procedure edits and returns the text string `s`. Characters that are written are echoed to the screen starting at the current position. Input operations are terminated by using any of the current termination keys in the current window.

At most `maxLen` characters can be edited, where `maxLen < SizeOf(s)`. Usually you call `RdStr` with

```pascal
(*$V-*)MGIO.RdStr(s,SizeOf(s) - 1);(*$V+*)
```

If proposal mode is *on*, the contents of the string `s` is written out before editing. The editing position is assigned to the end of the string. The cursor is shown if the cursor is *on*.

Characters are written out using the current window mode as the drawing mode, foreground and background colors, color mask etc. For input of characters and control codes, the current input procedure is called (as a default KbdInput).

If the scrolling mode is *on* and the editing is terminated using *Enter*, the current position moves to the beginning of the next text line (row).

The key code that is used for the termination key or combination of keys can be read by calling GetTermKey.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o
capability

Errors are also reported from StrEdit.
MGIO.SetProposal(TRUE);
MGIO.SetColRow(1,1);
s:= 'Proposal';
(*$V-*)MGIO.RdStr(s,SizeOf(s) - 1);(*$V+*)
MGIO.GetTermKey(key);

**RdWord**

\[ \text{P } \text{RdWord(VAR } c: \text{ Word } (*IO*)\); \]

This procedure edits and returns a non-negative integer in the variable parameter \( c \). Characters that are written are echoed to the screen starting at the current position. Input operations are terminated using any of the current termination keys in the current window.

The permitted characters are '0'..'9' and leading blanks. A maximum of 5 characters can be edited.

If the proposal mode is **on**, the contents in the variable \( c \) is written out before editing. The editing position is assigned to the end of the number. The cursor is shown if the cursor is **on**.

Characters are written out using the current window mode as the drawing mode, foreground and background colors, color mask etc. For input of characters and control codes, the current input procedure is called (as a default \text{KbdInput}).

If the scrolling mode is **on** and the editing is terminated using **Enter**, the current position moves to the beginning of the next text line (row).

The key code that is used for the termination key or combination of keys can be read using \text{GetTermKey}.

**SV.Result**

- **processWithoutWin**: the calling process has no window
- **capDontExist**: current window has no text i/o capability
- **invalidWord**: invalid integer

Errors are also reported from \text{StrEdit}. If an error occurs, **65535** is returned.
RdInt

P RdInt(VAR i: INTEGER (*IO*));

This procedure edits and returns an integer in the variable parameter i. Characters that are written are echoed to the screen starting at the current position. Input operations are terminated using any of the current termination keys in the current window.

The permitted characters are '+', '-', '0'..'9' and leading blanks. Plus and minus signs can only be written in the first position of the editing string. A maximum of 6 characters can be edited.

If the proposal mode is on, the number in the variable i is written out before editing. The editing position is assigned to the end of the number. The cursor is shown if the cursor is on.

Characters are written out using the current window mode as the drawing mode, foreground and background colors, color mask etc. For input of characters and control codes, the current input procedure is called (as a default KbdInput).

If the scrolling mode is on and the editing is terminated using Enter, the current position moves to the beginning of the next text line (row).

The key code that is used for the termination key or combination of keys can be read using GetTermKey.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
invalidInteger invalid integer

Errors are also reported from StrEdit. If an error occurs, MAXINT is returned.

MGIO.SetProposal(TRUE);
i:= 1;
MGIO.RdInt(i);
IF SV.Result <> SV.ok THEN ..
P RdLong(VAR li: LongInt (*IO*));

This procedure edits and returns a long integer in the variable parameter li. Characters that are written are echoed to the screen starting at the current position. Input operations are terminated using any of the current termination keys in the current window.

The permitted characters are '+', '-', '0'..'9' and leading blanks. Plus and minus signs can only be written in the first position of the editing string. A maximum of 12 characters can be edited.

If the proposal mode is on, the number in the variable li is written out before editing. The editing position is assigned to the end of the number. The cursor is shown if the cursor is on.

Characters are written out using the current window mode as the drawing mode, foreground and background colors, color mask etc. For input of characters and control codes, the current input procedure is called (as a default KbdInput).

If the scrolling mode is on and the editing is terminated using Enter, the current position moves to the beginning of the next text line (row).

The key code that is used for the termination key or combination of keys can be read using GetTermKey.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
invalidLongInt invalid long integer

Errors are also reported from StrEdit. If an error occurs, maxLongInt is returned.
RdReal

\begin{verbatim}
VAR r : Real; (*IO*)
digits: Word;
\end{verbatim}

This procedure edits and returns a floating-point number in the variable parameter \(r\). Characters that are written are echoed to the screen starting at the current position. Input operations are terminated using any of the current termination keys in the current window.

The permitted characters are '+', '-', ',', '0'..'9', 'E', 'e' and leading blanks. Plus and minus signs can only be written in the first position of the editing string and immediately after 'E' or 'e'. A maximum of 24 characters can be edited.

If the proposal mode is \textit{on}, the number in the variable \(r\) is written out, before editing, in \textit{exponent format} with the number of significant figures in the mantissa specified by the parameter \(\text{digits}\). The editing position is assigned to the end of the number. The value in \(\text{digits}\) is not important if the proposal mode is \textit{off}. The cursor is shown if the cursor is \textit{on}.

Characters are written out using the current window mode as the drawing mode, foreground and background colors, color mask etc. For input of characters and control codes, the current input procedure is called (as a default \texttt{KbdInput}).

If the scrolling mode is \textit{on} and the editing is terminated using \texttt{Enter}, the current position moves to the beginning of the next text line (row).

The key code that is used for the termination key or combination of keys can be read using \texttt{GetTermKey}.

\begin{verbatim}
SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
invalidReal invalid floating-point number
\end{verbatim}

Errors are also reported from \texttt{StrEdit}. If an error occurs, \textbf{maxReal} is returned.
MGIO.SetProposal(TRUE);
MGIO.SetCursor('>');
MGIO.SetCursorBlink(FALSE);
MGIO.RdReal(r,5);
IF SV.Result <> SV.ok THEN ..

**RdFixReal**

P RdFixReal(VAR r : Real;(*IO*)
          decimals: Word);

This procedure edits and returns a floating-point number in the variable parameter r. Characters that are written are echoed to the screen starting at the current position. Input operations are terminated using any of the current termination keys in the current window.

The permitted characters are '+', '-', '.', '0'..'9', 'E', 'e' and leading blanks. Plus and minus signs can only be written in the first position of the editing string and immediately after a 'E' or 'e'. A maximum of 24 characters can be edited.

If the proposal mode is on, the number in the variable r is written out, before editing, in decimal format with the number of decimals specified by the parameter decimals. The editing position is assigned to the end of the number. The value in decimals is not important if the proposal mode is off. The cursor is shown if the cursor is on.

Characters are written out using the current window mode as the drawing mode, foreground and background colors, color mask etc. For input of characters and control codes, the current input procedure is called (as a default KbdInput).

If the scrolling mode is on and the editing is terminated using Enter, the current position moves to the beginning of the next text line (row).

The key code that is used for the termination key or combination of keys can be read using GetTermKey.

SV.Result
processWithoutWin the calling process has no window
capDontExist current window has no text i/o capability
invalidReal invalid floating-point number
Errors are also reported from **StrEdit**. If an error occurs, **maxReal** is returned.

```pascal
MGIO.WrStr('Input the value of pi: ');  
MGIO.SetProposal(TRUE);  
pi:= 3.1416;  
MGIO.RdFixReal(pi,4);  
IF SV.Result <> SV.ok THEN
```

### GetTermKey

```pascal
P GetTermKey(VAR key: KbdCode (*0*));
```

This procedure returns, in the variable parameter **key**, the key code for the termination key or combination of keys, that were last used in the current window to interrupt or stop input operations for an "Rd" procedure.

The termination codes are assigned by the procedures **RdChar**, **RdStr**, **RdWord**, **RdInt**, **RdLong**, **RdReal** and **RdFixReal**.

```pascal
SV.Result  
processWithoutWin  
capDontExist  
```

- **processWithoutWin**: the calling process has no window
- **capDontExist**: current window has no text i/o capability

```pascal
MG.MatchForeground(MG.green);  
MGIO.WrStr('<Enter> accepts,  
          <Esc> return to standard file  <F1> help');  
MGIO.WrLn; MGIO.WrLn;  
MG.MatchForeground(MG.white);  
MGIO.WrStr('Name of file: ');  
MGIO.SetProposal(TRUE);  
MGIO.SetTermKeySet([[13,27,59]]);  
MG.MatchForeground(MG.yellow);  
s:= 'NONAME.TXT';
```
REPEAT
  MGIO.ClxCrEol;
  (*$V-*) MGIO.RdStr(s, SizeOf(s) - 1); (*$V+*)
  MGIO.GetTermKey(termKey);
  IF (termKey.wrd = MGIO.keys.fl.wrd) THEN BEGIN
    MG.GetWinMode(oldWinMode);
    MGIO.SetColRow(1, MGIO.CurWinRows);
    MG.MatchForeground(MG.green);
    MGIO.WrStr('At most eight characters, an optional period followed by at most three characters');
    MGIO.Wait(5000, break);
    MGIO.RdKbd(termKey);
    MGIO.SetColRow(1, MGIO.CurWinRows);
    MGIO.ClxCrEol;
    MG.SetWinMode(oldWinMode);
  END
  ELSE IF (termKey.wrd = MGIO.keys.esc1.wrd) THEN
    s := 'NONAME.TXT';
UNTIL termKey.wrd = MGIO.keys.cr.wrd;

The example shows how you can incorporate help and undo functions using GetTermKey, when reading text input.
MGDL - Display Lists

The units MGDL and MGDL87, in cooperation with MG, implement a technique for saving graphics commands with operations and data (recording) in display lists. If requested, the saved commands can be modified (editing) and later repeated once again (playback). Recorded graphics commands can be saved in files on disk and then reloaded and played back.

Display lists can be used just like notebooks for graphics commands. They can, for instance, be used to remember and restore the graphics in a window that has been hidden. Display lists can also be used to move graphics between windows, screens and devices. You may use display lists to store graphics that are created interactively on the screen and then print the result using a higher resolution on a laser or ink jet printer. In all cases, display lists are a good alternative in comparison with saving and moving graphics by using virtual screens or images in raster format (as pixels).

Display lists can also be used to automatically scale drawn graphics or to save drawing commands permanently on disk instead of hard coding them into the application.

Recorded graphics commands are stored in objects of the type MG.DisplayList (of pointer type). The internal structure of display lists may differ, but usually they are implemented as arrays, linked lists or files.

MultiGraphics does not have any specific demands on the internal structure of display lists, except that a display list must implement an Append procedure that makes it possible for drawing procedures, such as MG.DrawLineTo, to add (append) new display elements to the list. Using the Append procedure, a display list can also be updated directly, instead of executing direct drawing commands.
The MGDL unit implements display lists using the generic linked list unit DBList, which is supplied with MultiGraphics.

All window related drawing and status modifying graphics commands can be stored in display lists. Query or reading commands cannot. Examples of valid display list commands are MG.DrawLine, MG.DrawPoint, MG.MatchForeground, MGIO.WrInt, MGIO.SetColRow, MGImg.DrawImage, MGW.FillPolygon, MGIW.DrawStr, etc. Examples of query commands are MG.CurX, MG.CurForeground, MGIO.RdInt, MGImg.ReadImage, etc., which cannot be stored in display lists (they are ignored during recording).

Display lists in MultiGraphics are window oriented, which means that each active window can have its own unique display list. The recording function is activated separately for each window, which means that recording may be active in some windows, but switched off in others.

This also means that only graphics commands that perform specific window operations may be stored, consequently the following commands cannot; commands for changing windows (e.g. MG.SetWin), screen oriented commands (e.g. MG.SetScr) or device oriented commands (e.g. MG.SetPalette). However, there are a few exceptions from this strict rule, e.g. MG.ZoomVirtScr and MGImg.ZoomImage.

Recorded display lists are not, on the other hand, restricted to being played back in the same window that they were recorded in, but can be played back in other windows on the same screen, in windows on other screens and even in windows on other devices.

Display lists can be saved in files on disk (file type ".DL" is recommended) and then be loaded again. Display files are portable between all programming languages and compilers that are supported by MultiGraphics. A display file produced by, for example, a Borland Pascal program can be used in programs compiled by StonyBrook Pascal+, TopSpeed C/C++, TopSpeed Modula-2, and so on.

The MGDO unit contains an extended technique for handling libraries with graphics objects based on display lists.
Recording Display Commands

You can create a display list quite simply by recording graphics commands at the same time as they are executed.

However, first you have to create a new and empty display list. You can do this by calling `MGDL.CreateDisplayList`. The display list is attached to a variable of the type `MG.DisplayList`. The recording of graphics in the current window is then activated by calling `MG.StartRecording` and it continues until `MG.StopRecording` is called. The recording can later be activated again with a new call to `MG.StartRecording`, in which case the display list is appended with new graphics commands. Here is an example:

```pascal
VAR
dl: MG.DisplayList;

MGDL.CreateDisplayList(dl);
MG.SetWin(win);
MG.StartRecording;
MGIW.SetPos(5000,5000); R
MGIW.DrawStr('Recording in progress'); R
MGImg.LoadPCXImage(img,'CLAPPER.PCX');
MGIW.SetRelPos(0,1000); R
MGImg.DrawImage(img); R
MGImg.DisposeImage(img);
MG.StopRecording;

MGDL.DisposeDisplayList(dl);
```

Any graphics commands stored in the display list are marked with an "R".

The recorded display list, which is on the program heap, contains a series of display elements formed in accordance with the generic structure `MG.DisplayElement`. Each recorded display procedure is specified using a unique number (see units `MGDP.IFC`, `MGIWDP.IFC`, `MGWDP.IFC/MGWDP87.IFC`, `MGPatDP.IFC`, `MGImgDP.IFC` and `MGIODP.IFC/MGIODP87.IFC` where these numbers are defined). Procedure parameter data, which in scope and structure differ between various display commands, are also stored in the display elements.

Nothing prevents you from recording in several windows simultaneously. Normally each window has its own display list, but it is not against the rules to use the same display list (if you can find a use for this).
Recording at the Highest Level

It is common that display commands at a higher level, e.g. routines in the units MGW, MGIW, MGIO and MGImg, make calls to MultiGraphics commands at lower level (often in MG). The recording of display commands is always carried out at the "highest" level, i.e. only those graphics commands that are called directly by the application program are stored in the display list - not the internal calls by graphics commands in MultiGraphics.

Playback

A display list can be played back in the same window that it was created in, or in another window on the same screen, a window in another screen or even in a window on another device!

In order to execute the graphics commands in a display list, it is necessary to link all graphics procedures that are needed during playback. Since it is difficult to predict beforehand exactly what commands that are to be found in a display list, the linking is done per unit basis, i.e. all "displayable" procedures in a specific unit are linked in their entirety (Borland Pascal/Turbo Pascal's optimized linking normally only links those procedures that are directly called in the application source). The linking is carried out by calling the Link procedure in each respective unit, i.e. MG.Link, MGPat.Link, MGIW.Link, MGW.Link, MGImg.Link and MGIO.Link.

The next step is to select a window for the playback and verify that this window has the necessary enhanced capabilities (e.g. for world coordinates, i/o, etc). Then set those window attributes that are not explicitly set by the display list commands. Perhaps you have not recorded the selection of foreground color or current font in the display list, but instead want to keep this option open until you start the playback. You can then assign the desired foreground color and font just before pressing the "Play" button.

Now, you need to set the desired starting position in the display list, i.e. the position for the first display command that is to be played back. Usually this is the first position in the list (set by calling MGDL.SetFirstDisplayListPos), but may also be an element somewhere in the middle of the list (MGDL.SetDisplayListPos).
To play back the current graphics command in the display list `MGDL.DrawDisplayElement` is to be called. You can move to the next element with `MGDL.NextDisplayListPos`. By calling `MGDL.DrawDisplayList` the rest of the list is played back starting from the current list position.

The following example shows how the complete display list can be played back in the same window as the recording was done (refer to the example above):

```pascal
MG.Link; MGIW.Link; MGImg.Link; MGIO.Link;
  ..
MG.EnhanceWin(win,MGIW.cap);
MG.EnhanceWin(win,MGIO.cap);
  ..
MG.SetWin(win);
MG.ClrWin;
MGDL.SetFirstDisplayListPos(dl);
MGDL.DrawDisplayList(dl);
  ..
```

The same display list can be played back in another window with another foreground color and font:

```pascal
MG.EnhanceWin(win1,MGIW.cap);
MG.EnhanceWin(win1,MGIO.cap);
MG.SetWin(win1);
MG.MatchForeground(MG.green);
MG.LoadFont(fnt,'ROMAN.COL',TRUE);
MG.SetFont(fnt);
MGDL.SetFirstDisplayListPos(dl);
MGDL.DrawDisplayList(dl);
```

Here is how the first two display commands in the display list can be played back:

```pascal
MGDL.SetFirstDisplayListPos(dl);
FOR i:= 0 TO 1 DO BEGIN
  MGDL.DrawDisplayElement(dl);
  MGDL.NextDisplayListPos(dl);
END; (*for*)
```

Since a large amount of code is linked to the application when playing back display lists, it may be worth considering placing the playback section in a separate program, which is started using `DOS.Exec`.
Processing Display Lists

You can add to, modify and delete display commands in display lists. Display lists can also be created without recording.

**MGDL.DeleteDisplayElement** will delete the current element in the display list. In the next example, we will show you how to delete the last two graphics commands in the example above and then play back the rest of the list:

```pascal
MGDL.SetFirstDisplayListPos(dl);
FOR i:= 0 TO 1 DO BEGIN
    MGDL.NextDisplayListPos(dl);
END; (*for*)
MGDL.DeleteDisplayElement(dl);
MGDL.DeleteDisplayElement(dl);
MGDL.SetFirstDisplayListPos(dl);
MGDL.DrawDisplayList(dl);
```

The data for a graphics commands in the display lists can be accessed by calling **MGDL.GetDisplayElementPtr**, which makes it possible to inspect and - with caution - also modify data. The numbers and data structures for display commands are to be found in the "DP" units ("Display Procedures"), i.e. **MGDP**, **MGPatDP**, **MGIWDP**, etc. In the next example, we have decided only to draw the image (we assume that the original display list is used):

```pascal
VAR
dePtr: MG.DisplayElementPtr; (* Generic element type *)
.. MGDL.SetFirstDisplayListPos(dl);
WHILE MGDL.CurDisplayListPos(dl) <> MGDL.EndDisplayListPos(dl) DO BEGIN
    MGDL.GetDisplayElementPtr(dl,dePtr);
    IF dePtr^.displayProc = MGImgDP.dpDrawImage THEN
        MGDL.DrawDisplayElement(dl);
        MGDL.NextDisplayListPos(dl);
END; (*while*)
```

Finally, we will expand the display list with a command that assigns the foreground color before the text string "Recording in progress" is written out:
VAR
dePtr : MGDP.MGDisplayElementPtr;
deSize: Word;
..
deSize:= 2*SizeOf(Word) (* first two fields *)
  + SizeOf(MGDP.MatchForegroundParaType);
GetMem(dePtr,deSize);
WITH dePtr^ DO BEGIN
  paraSize:= deSize;
  displayProc:= MGDP.dpMatchForeground;
  paraMatchForeground.rgbColor:= MG.yellow;
END; (*with*)
MGDL.SetFirstDisplayListPos(dl);
MGDL.InsertDisplayElement (dl,
  MG.DisplayElementPtr(dePtr));
FreeMem(dePtr,deSize);
MGDL.SetFirstDisplayListPos(dl);
MGDL.DrawDisplayList(dl);

Display Files

Display lists can be saved permanently on disk in display files. The recommended file type is "DL". Here is how it works:

First, a new display file is created (if it has not been created before) using MGDL.CreateDisplayFile. This file is then expanded by calling MGDL.AppendToDisplayFile, in which case new graphics commands are appended to the file starting from the current position in the display list. Still using the example above:

MGDL.CreateDisplayFile('TEST.DL');
MGDL.SetFirstDisplayListPos(dl);
MGDL.AppendToDisplayFile (dl,maxLongInt,'TEST.DL');

Here, all elements in the display list are saved.

Commands from display files can be loaded again and added to an existing display list in memory. MGDL.AppendFromDisplayFile appends commands from the display file to the display list in memory, while MGDL.InsertFromDisplayFile inserts the commands before the current position in the display list.

MGDL.CreateDisplayList(dl1);
MGDL.AppendFromDisplayFile (dl1,'TEST.DL',0,maxLongInt);
MGDL.SetFirstDisplayListPos(dl1);
MGDL.DrawDisplayList (dl1);
MGDL.DisposeDisplayList (dl1);
Here, the complete display list is loaded into a new display list variable, \texttt{dl1}.

Display files are portable between all programming languages and compilers that are supported by \textbf{MultiGraphics}.

**Value or Reference**

Most graphics procedures use parameters for transferring data. Data can, however, be transferred in \textit{two} ways, namely by

```
" value or
" reference.
```

\texttt{MG.SetPos(x,y)} and \texttt{MG.DrawStr(s)} give two examples of transferring data by \textit{values} which, when recorded, will store the actual values of \texttt{x}, \texttt{y} and \texttt{s} directly in the display list. Procedures that transfer data by values are labeled "DL" ("Display List") in their headings.

However, some procedures transfer data by \textit{reference} (i.e. addresses or pointers) instead of direct values. This applies to, for example, \texttt{MG.SetPattern}, \texttt{MG.SetSymbols}, \texttt{MG.SetFont}, \texttt{MG.CopyRect}, \texttt{MGImg.DrawImage}, etc. These procedure's are labeled "DLR" in their headings, where "R" stands for \textbf{reference}.

In general, what should be remembered is that references must be valid, i.e. data must still be kept in memory at the same addresses, when a display list with references is played back. This is especially important to remember if you intend to save display lists permanently on disk for later use.

\texttt{MG.CopyRect}, for example, has a window as its first parameter, and this window must still be there when playing back. When saving display lists on disk and reloading them, this is only achievable during the same program session and presumes, in our example, that the window referred to is still there at the same address! In a later session of the program, the chances that the window data will be exactly at the same memory address are almost non-existent and any attempt to play it back will cause \texttt{MG.CopyRect} to return an error message.

Wherever possible, you should avoid procedures that use references when your aim is to save display lists permanently. When dealing with line styles and patterns, we recommend using \texttt{MGPat.SetGenLineStyle} and \texttt{MGPat.SetGenPattern}, which handle generic line styles and patterns.
without memory references, instead of `MG.SetLineStyle` and `MG.SetPattern`, which contain references.

When recording the procedures `MG.SetSymbols`, `MG.SetFont`, `MGImg.DrawImage`, `MGImg.DrawImagePart` and `MGImg.ZoomImage`, the display list elements will also have information about the file names of the symbol libraries, fonts and images. Instead of using memory references, symbols, fonts and images are normally loaded again from disk. These procedures are labelled "DL(R)" in their headings. An example will make this clearer; this is how it works using `MG.SetFont`:

The parameter `fnt` in `MG.SetFont(fnt)` contains the reference to an internal and quite complicated font structure. In the display list, the reference and file name of the font file are stored, but not the internal font structure (which would perhaps require tens of kilobytes).

If we, when reading graphics commands from a display file to a display list in memory, come across the command `MG.SetFont` in the file, the specified font is normally loaded again from its font file on disk, even if the font is already in memory.

By calling `MGDL.ResetLoader(TRUE)`, before reading from disk, you can instruct `MGDL` first of all to test if the old font reference is still valid, and in that case re-use it; otherwise the font will be reloaded from disk.

**Portable Display Lists**

Display lists can be made more or less "portable". If, for example, you do not want an object drawn to be locked to a particular color, the solution is to omit the color selection in the display list. The desired color is instead set before playback.

If you do not want an object to be confined to a particular drawing position, you omit setting absolute positions in the display list and instead use relative positions (the position can also be moved by placing the playback window at another position than the recording window). Here are some more recommendations to get higher portability:

* Use world coordinates (`MGIW` or `MGW/MGW87`) in order to adapt the size to windows and resolutions of different screens and devices. Using world coordinates, you can also enlarge and reduce graphics by simply changing the world coordinates in the playback window.
Use relative positions instead of absolute, if you want to save "display objects" that will be simple to move.

Use `MG.MatchForeground` and `MG.MatchBackground` instead of device specific colors. This makes it easier to move graphics between different devices and between color and monochrome screens.

Use generic line styles and patterns in `MGPat`.

**MGDL versus MGDL87**

The unit **MGDL** is used when you build on Borland Pascal/Turbo Pascal's `REAL` type. **MGDL87** is used when you want to utilize the standard 8-bytes `Double`, which is especially suitable for PCs with math processors, 80x87 or 486DX. In display files, floating point data is always stored in the `Double` format which allows the same display files to be used by both **MGDL** and **MGDL87**. They can also be used by programs compiled by other compilers (e.g. TopSpeed) which support the standard `Double` format.

**Error Handling**

The unit **MGDL** uses the `SuperVisor` technique for error handling. Errors are automatically taken care of by error procedures (as a default `SV.GlbEP` is called) or manually by checking `SV.Result` after calling procedures and functions in **MGDL**.

**Examples**

**GADL** shows recording and playback of display lists using both pixel coordinates as well as floating-point based world coordinates. **GAIDL** is the same example, but using integer based world coordinates instead. If you have more than one screen connected to your computer (e.g. VGA + Hercules), the playback can be carried out on both screens (with a third 8514/A screen, you can even do it on three (!) screens). The display list, moreover, is saved and loaded from a display file.
Constant in MG for Display Lists

\texttt{MG.nullDisplayList = MG.DisplayList(NIL);}

Constant to indicate a non-existent display list.

Data Types in MG for Display Lists

The following data types are defined in the unit \texttt{MG} and are independent of the specific technique for implementation of display lists.

\textbf{MG.DisplayList}

Abstract data types for display lists. Each active display list must have its own variable of this type.

\texttt{MG.DisplayElement = RECORD}
\begin{verbatim}
  paramsSize : Word;
  displayProc: Word;
  params     : ARRAY[0..0] OF Byte;
END;
\end{verbatim}

Generic data type which serves as a template for elements in display lists. The first two fields are always the same - \texttt{paramsSize} specifies the total size of the element's structure while \texttt{displayProc} specifies a unique number for the element's display procedure. \texttt{params}, whose real size depends on the display command, contains procedure parameters. Each display command has its own variant of this basic type - see the "DP" units \texttt{MGDP}, \texttt{MGIWDP}, etc. for the declarations.

\texttt{MG.DisplayElementPtr = ^MG.DisplayElement;}

Pointer to \texttt{MG.DisplayElement}. 
Procedures & Functions in MG for Display Lists

**MG.StartRecording**

P StartRecording(dl: DisplayList);

Starts the recording of graphics commands in the current window and stores all recorded commands in the display list dl. Prior to this, the display list dl must have been created using MGDL.CreateDisplayList. All MultiGraphics procedures executed and labeled "DL.." in their headings will be recorded.

dl becomes the current display list in the current window. dl need not be empty when the call is made, since display commands are appended to the display list.

**SV.Result**

MGErr.processWithoutWin the calling process has no window
MGErr.invalidDisplayList invalid display list in dl

VAR
dl : MG.DisplayList;
win: MG.Window;
.
MGDL.CreateDisplayList(dl);
MG.SetWin(win);
MG.StartRecording(dl);
.
MG.StopRecording;

**MG.StopRecording**

P StopRecording;

Stops the recording of graphics commands in the current window. The recording can be restarted by calling StartRecording again.

**SV.Result**

MGErr.processWithoutWin the calling process has no window
MG.CurDisplayList

P CurDisplayList: DisplayList;

Returns the current display list in the current window. If no display list is activated in the window, or when there is an error, MG.nullDisplayList is returned.

SV.Result
MGErr.processWithoutWin the calling process has no window

Link Procedures

MG.Link, MGPat.Link, MGIW.Link, MGW.Link, MGImg.Link, MGIO.Link

P MG.Link;
P MGPat.Link;
P MGIW.Link;
P MGW.Link;
P MGImg.Link;
P MGIO.Link;

Links all "displayable" graphics procedures in the units MG, MGPat, MGIW, MGW, MGImg and MGIO so that display commands from these units can be executed during playback (MGDL.DrawDisplayElement and MGDL.DrawDisplayList).

Before you start to play back a display list, all units that contain graphics display commands and that are present in the display list, must be linked using the associated Link procedure.

We have chosen this technique for the sake of size optimization. A display list can contain anything from a single graphics command to everything in a unit or graphics commands from different units. The exact composition of graphics commands in a display list is in general not known at compile time. For this reason, the code for all potential graphical procedures must be linked, in case they show up in the display list.

In order not to link superfluous code, e.g. when the program does not play back display lists, the linking is done manually per unit, by calling each respective Link procedure. If it is likely that a display list may contain procedures from all units in
MultiGraphics, all Link procedures must be called, otherwise you should restrict yourself to those units that you know do occur. Notice that the Link procedures only needs to be called before playback - not when recording.

SV.Result
MGErr.displayArrayFull the internal array with
drawing units is full

Constants in MGDL

unitName 'MGDL'
copyright 'DATABITEN'

invalidDisplayListFiley 1500
Specific error code in MGDL.

nullDisplayListPos = DisplayListPos NIL;
Constant to indicate a non-existent position in display lists.

Data Types in MGDL

DisplayListPos
Abstract data type for positions in display lists.
Variables in MGDL

revision : Word;
EP : SV.ErrProc;

Revision number and local error handling procedure for this unit.

Procedures & Functions in MGDL

The Display List as a Whole

CreateDisplayList

P CreateDisplayList(VAR dl: MG.DisplayList (*O*));

Creates a new, empty display list attached to dl. Must be called before recording, direct processing or loading display list commands from disk.

The new display list is compatible with lists in DBList.

SV.Result
MGErr.outOfMem not enough memory

Also error codes from DBList.

ClearDisplayList

P ClearDisplayList(dl: MG.DisplayList (*Io*));

Empties the display list dl of display commands, but dl can be used again without calling CreateDisplayList.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.
DisposeDisplayList

P DisposeDisplayList(VAR dl: MG.DisplayList (*IO*));

Deletes the display list attached to dl from memory. dl is set to MG.nullDisplayList.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.

DisplayListOK

F DisplayListOK(dl: MG.DisplayList):BOOLEAN;

Returns TRUE if dl is attached to a valid display list, otherwise FALSE.

DisplayElements

F DisplayElements(dl: MG.DisplayList): Word;

Returns the current number of display elements (graphics commands) in the display list dl.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.

Processing Display Lists

AppendDisplayElement

P AppendDisplayElement(dl : MG.DisplayList;(*Io*)
  dePtr: MG.DisplayElementPtr);

Appends a new display element (graphics command), addressed by dePtr, to the end of the display list dl. The current position is set to the end of the list. dePtr points to data of the structure MG.DisplayElement.
With this procedure and **InsertDisplayElement**, display lists can be filled without recording.

**SV.Result**
**MGErr.invalidDisplayList** invalid display list

Also error codes from **DBList**.

VAR

```
dl : MG.DisplayList;
dePtr : MGWDP.MGWDisplayElementPtr;
deSize : Word;
p : ARRAY[0..2] OF MGW.Point;
```

```
deSize:= 2*SizeOf(Word) (* first two fields *)
  + SizeOf(MGWDP.DrawLineParaType);
GetMem(dePtr,deSize);
WITH dePtr^ DO BEGIN
  paraSize:= deSize;
  displayProc:= MGWDP.dpDrawLine;
  paraDrawLine.dist:= 0.5;
END; (*with*)
MGDL.AppendDisplayElement(dl,
                         MG.DisplayElementPtr(dePtr));
FreeMem(dePtr,deSize);
```

```
deSize:= 2*SizeOf(Word) (* first two fields *)
  + SizeOf(MGWDP.DrawPolyLineParaType)
  + 2*SizeOf(MGW.Point); (* number of points- 1*)
GetMem(dePtr,deSize);
WITH dePtr^ DO BEGIN
  paraSize:= deSize;
  displayProc:= MGWDP.dpDrawPolyline;
  paraDrawPolyline.first:= 0;
  paraDrawPolyline.last:= 2;
  FOR i:= 0 TO 2 DO
    paraDrawPolyline.polyline[i]:= p[i];
END;
END; (*with*)
MGDL.AppendDisplayElement(dl,
                         MG.DisplayElementPtr(dePtr));
FreeMem(dePtr,deSize);
**InsertDisplayElement**

```pascal
P InsertDisplayElement(dl : MG.DisplayList;(*Io*)
                  dePtr: MG.DisplayElementPtr);
```

Inserts a new display element (graphics command), addressed by `dePtr`, in the display list `dl before` the current position. `dePtr` points to data of the structure `MG.DisplayElement`.

With this procedure and **AppendDisplayElement**, display lists can be produced without recording.

**SV.Result**

- **MGErr.invalidDisplayList** invalid display list

Also error codes from **DBList**.

**DeleteDisplayElement**

```pascal
P DeleteDisplayElement(dl: MG.DisplayList (*Io*));
```

Deletes the current element in the display list `dl`.

**SV.Result**

- **MGErr.invalidDisplayList** invalid display list

Also error codes from **DBList**.

**GetDisplayElementPtr**

```pascal
P GetDisplayElementPtr(    dl   : MG.DisplayList;
                      VAR dePtr: MG.DisplayElementPtr);
```

Returns the address in the pointer `dePtr` to the current element in the display list. `dePtr` points to a data area of `MG.DisplayElement` structure.

This procedure can be used to read, and if requested, to modify display elements (graphics commands) in display lists.

**SV.Result**

- **MGErr.invalidDisplayList** invalid display list

Also error codes from **DBList**.
Playback

**DrawDisplayElement**

```pascal
P DrawDisplayElement(dl: MG.DisplayList);
```

Executes the graphics command that is stored in the current display element of the display list `dl`. The current position in the list is *not* changed.

The graphics procedure that is called must be linked to the application, using the `Link` procedure in the unit that exports the requested procedure. Otherwise an error message is returned.

If the graphics command uses a specific capability, the current window must be enhanced with this capability. If the display element contains, for example, a procedure from `MGIW`, `MGIW.cap` must be installed in the window where the display command is played back (see `MG.EnhanceWin` and `MG.EnhanceScr`).

**SV.Result**

- `MGErr.processWithoutWin` the calling process has no window
- `MGErr.invalidDisplayList` invalid display list
- `MGErr.displayProcOutOfRange` the number of the display procedure is outside the range of linked units
- `MGErr.invalidDisplayProc` the graphics display procedure not available or not linked

**DrawDisplayList**

```pascal
P DrawDisplayList(dl: MG.DisplayList (*Io*));
```

Executes all the graphics commands in the display list `dl starting from the current position` to the end of the list. The new current position is set to the end of the list. The display operations are carried out in the current window.

All graphics procedures that are called by the display commands in the display list must be linked with the `Link` procedure of each respective unit. Otherwise an error message is returned.

The display list need not be recorded in the same window as it is played back in, but the playback window must be enhanced with any necessary
capabilities. If the display list, for example, contains graphics commands from MGIW, MGIW.cap must be installed in the window where the display list is played back (see MG.EnhanceWin and MG.EnhanceScr).

SV.Result
MGErr.processWithoutWin calling process has no window
MGErr.invalidDisplayList invalid display list
MGErr.displayProcOutOfRange number of the display procedure outside the range of the linked units
MGErr.invalidDisplayProc the display procedure not available or not linked

Positions in Display list

**SetDisplayListPos**

```pascal
P SetDisplayListPos(dl : MG.DisplayList (*Io*); pos: DisplayListPos);
```

Sets the new current position, `pos`, in the display list `dl`. The position `pos`, has normally received its value from one of the following functions; FirstDisplayListPos, LastDisplayListPos or CurDisplayListPos.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.

**NextDisplayListPos**

```pascal
P NextDisplayListPos(dl: MG.DisplayList (*Io*));
```

Moves the current position forward one element in the display list `dl`.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.
PrevDisplayListPos

P PrevDisplayListPos(dl: MG.DisplayList (*Io*));

Moves the current position back one element in the display list dl.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.

SetFirstDisplayListPos

P SetFirstDisplayListPos(dl: MG.DisplayList (*Io*));

Moves the current position to the first display element in the display list dl.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.

FirstDisplayListPos

F FirstDisplayListPos(dl: MG.DisplayList): DisplayListPos;

Returns the position of the first display element in the display list dl.

When an error occurs nullDisplayListPos is returned.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.

EndDisplayListPos

F EndDisplayListPos(dl: MG.DisplayList): DisplayListPos;

Returns the position for the last display element in the display list dl.

When an error occurs nullDisplayListPos is returned.
SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.

WHILE MGDL.CurDisplayListPos(dl) <> MGDL.EndDisplayListPos(dl) DO BEGIN
  MGDL.DrawDisplayElement(dl);
  MGDL.NextDisplayListPos(dl);
END; (*while*)

**CurDisplayListPos**

F CurDisplayListPos(dl: MG.DisplayList): DisplayListPos;

Returns the current position in the display list dl.

When an error occurs nullDisplayListPos is returned.

SV.Result
MGErr.invalidDisplayList invalid display list

Also error codes from DBList.

**Display Files**

**CreateDisplayFile**

P CreateDisplayFile(fname: String);

Creates a new and empty display file with file name and path in fname. The file can then be expanded with graphics display commands by calling AppendToDisplayFile.

Error codes from DBList.

MGDL.CreateDisplayFile('C:\mg\CHART.DL');
AppendToDisplayFile

P AppendToDisplayFile(dl : MG.DisplayList;
elements : Word;
fname : String);

Augments the display file fname with a number of elements graphics display commands, starting from the current position in the display list dl. The display file must exist prior to this (a new and empty display file is created by calling CreateDisplayFile). The new display commands are appended to the file.

Display commands are written from the current position in dl and with a maximum of elements element. If elements > the number of remaining elements in the display list, the rest of the elements are saved (elements can be set to maxLongInt). The current position in the display list is not modified.

SV.Result
MGErr.invalidDisplayList invalid display list
invalidDisplayListFile not a display file

Also error codes from DBList.

MGDL.SetFirstDisplayListPos(dl);
MGDL.AppendToDisplayFile(dl,
    maxLongInt,
    'C:\mg\CHART.DL');

ResetLoader

P ResetLoader(reuse: BOOLEAN);

Selects the mode for how symbols, fonts and images are to be loaded when reading from display files using AppendFromDisplayFile and InsertFromDisplayFile.

With reuse = FALSE (default setting) symbols, fonts and images are always loaded again from files on disk. This guarantees that the correct objects will be placed in the display list, but this can be wasteful with memory if the object is already present in memory.
With `reuse = TRUE`, MGDL first examines the reference data in the display element, to see if the symbols, fonts or images are already loaded into memory. If the references are still valid, MGDL will reuse them; otherwise it will try to load the symbols, fonts and images from files on disk.

There is a possibility that an existing reference will be valid, but that the data pointed at has been modified after the display list was saved. It is safer therefore to use `reuse = FALSE`.

Symbol, font and image files to be loaded must be in the current directory. N.B. that data for symbols, fonts and images are never stored directly in the display files, but in separate symbol, font and image files. Objects are loaded according to the normal rules for `MG.LoadSymbols`, `MG.LoadFont`, `MGImg.LoadImage` and `MGImg.LoadPCXImage`.

**AppendFromDisplayFile**

```pascal
procedure AppendFromDisplayFile (dl : MG.DisplayList;
                               fname : String;
                               startElement : LongInt;
                               elements : Word);
```

Augments the display list `dl` with commands that are loaded from the display file with the name and path in `fname`. A number of `elements` display commands, starting with the element `startElement` (the first element in the file has the number 0), are loaded from the display file and appended to `dl`. The new current position in `dl` is set to the end of the augmented list. The display list must have been created previously (a new, empty display list is created with `MGDL.CreateDisplayList`).

Display elements in the display file are numbered from 0 and upwards. If `elements` are given the value `maxLongInt`, the rest of the elements in the display file are loaded.

Display commands that have references to symbols, fonts or images are normally loaded again from the appropriate symbol, font and image files on disk. By calling `ResetLoader(TRUE)` the loading technique is changed so that `AppendFromDisplayFile` first checks if the object is already in memory. If so, they are not loaded again from disk. `ResetLoader` must be called before loading.

**SV.Result**

- `MGErr.invalidDisplayList` invalid display list
MGDL.invalidDisplayListFile not a display file

Also error codes from DBList.

MGDL.AppendFromDisplayFile(dl, 'C:\mg\CHART.DL', 5,10);

InsertFromDisplayFile

P InsertFromDisplayFile(dl : MG.DisplayList;
fname : String;
startElement : LongInt;
elements : Word);

Augments the display list dl with commands that are loaded from the display file with the name and path in fname. A number of elements display commands, starting with the element startElement (the first element in the file has the number 0), are loaded from the display file and inserted before the current element in dl. The display list must have been created previously (a new, empty display list is created with MGDL.CreateDisplayList).

Display elements in the display file are numbered from 0 and upwards. If elements are given the value maxLongInt, the rest of the elements in the display file are loaded.

Display commands that have references to symbols, fonts or images are normally loaded again from the appropriate symbol, font and image files on disk. By calling ResetLoader(TRUE) the loading technique is changed so that AppendFromDisplayFile first checks if the object is already in memory. If so, they are not loaded again from disk. ResetLoader must be called before loading.

SV.Result
MGErr.invalidDisplayList invalid display list
MGDL.invalidDisplayListFile not a display file

Also error codes from DBList.
Using display lists (see "MGDL - Display Lists"), you can save graphics commands in order to "play them back" later. By a display object we denote a graphic object with an individual name and number and with the graphics information stored in a display list. A display object is always included in a larger collection of display objects, this is known as a display object library. The name and number of a display object is unique within the display object library but can coincide with names and numbers of display objects in other libraries.

The unit MGDO handles both the display object library as a whole, and the individual display objects in the library. Display object libraries can be stored permanently in files on disk (the file type ".DO" is recommended). Display objects from such libraries can then be loaded and "played back" again in program after program. MGDO makes it easier to build up a library of useful, general graphics objects as opposed to handling and storing all the objects individually with their own display files.

Display object libraries are handled as variables of the type Display-Objects. Individual display objects are described by DisplayObjectInfo structures.

You can think of display object libraries as a generalization of symbol libraries (the MG.Symbols type). Symbols are always monochrome and consist of bit blocks or polylines or filled polygons or filled/unfilled Bézier curves. Display objects can contain all types of graphics commands in an object, and the same object can mix
colors freely and moreover contain circles, arcs, sectors, text, symbols, images, etc. All the graphics commands that can be stored in display lists (labeled with "DL" in their headings) may also be included in display objects. It is these unlimited possibilities that make libraries of display objects so useful.

Display object libraries are portable between different programming languages and compilers. Objects that have been produced by for example Borland Pascal may therefore be used by programs compiled with Stony-Brook Pascal+, TopSpeed C/C++, TopSpeed Modula-2, and so on.

Creating a Library of Display Objects

It is simple to build a display object library and then save it permanently on disk. Here are the steps to follow:

1. Create a new, empty display object library by calling `CreateDisplayObjects`.

2. Create a display list with drawing commands for each display object that will be included in the library. The simplest way to do this is by recording, but the display list can also be filled "manually".

3. Give a unique number (within the range 0..127) and a name (max. 12 characters) for each display object and add object by object to the library using `AddDisplayObject`. A display object library can hold a maximum of 128 display objects.

4. Finally, save the display object library permanently on disk with `SaveDisplayObjects`.

The display object library, and the display lists it uses, may now be deleted from memory using `DisposeDisplayObjects`.

```pascal
VAR
do: MGDO.DisplayObjects;
dl: ARRAY[0..29] OF MG.DisplayList;
.. MGDO.CreateDisplayObjects(do);
FOR i:= 0 TO 29 DO BEGIN
   MGDL.CreateDisplayList(dl[i]);
END;
.. MG.StartRecording(dl[0]);
.. The graphics commands for objects 0
```
MG.StopRecording;
MGDO.\texttt{AddDisplayObject}\,(do,0,dl[0],\texttt{\textquotesingle}Adam\textquotesingle);

MG.StartRecording(dl[1]);
.. The graphics commands for objects 1
MG.StopRecording;
MGDO.\texttt{AddDisplayObject}\,(do,1,dl[1],\texttt{\textquotesingle}Bert\textquotesingle);
.. MG.StartRecording(dl[29]);
.. The graphics commands for objects 29
MG.StopRecording;
MGDO.\texttt{AddDisplayObject}\,(do,29,dl[29],\texttt{\textquotesingle}Carl\textquotesingle);

MGDO.\texttt{SaveDisplayObjects}\,(do,\texttt{\textquotesingle}NAMES.DO\textquotesingle);
MGDO.\texttt{DisposeDisplayObjects}\,(do);
FOR \texttt{i:= 0 TO 29 BEGIN}
\hspace{1em} MGDL.DisposeDisplayList(dl[i]);
\texttt{END;}

\textbf{Draw Display Objects}

It is just as easy to use display objects:

1. Link the graphics commands from those units that may be used by the display object. This is done by calling \texttt{MG.Link}, \texttt{MGIW.Link}, etc. (see "MGDL - Display Lists" for more information).

2. Install all enhanced window capabilities that are required by the display objects.

3. Load all the desired display objects from a display object file with \texttt{LoadDisplayObjects}. The desired display objects are specified by listing their object numbers. Or alternatively, you may load the objects using \texttt{LoadNamedDisplayObjects} by listing the names of the desired objects. In memory you will now have a display object library that comprises some or all of the display objects in the display object file.

4. If the display objects have been loaded based on their names, you must now find out their object numbers. This can be done with \texttt{DisplayObjectId}.

5. Select the current window. Display objects are drawn in the current window with \texttt{DrawDisplayObject}. The desired objects are specified using their object numbers.
6. The display object library in memory, including the display object's display lists, can be deleted using **DisposeDisplayObjects**.

```pascal
VAR
do1, do2: MGDO.DisplayObjects;
...
MG. Link; MGIW. Link; MGIO. Link; MGImg. Link; ...
...
MG. EnhanceWin(win, MGIW.cap);
MG. EnhanceWin(win, MGIO.cap);
...
MGDO. LoadDisplayObjects (do1, 'NAMES.DO',
[0..9, 20, 25..29]);
MG. SetWin(win);
MGIW. SetPos(5000, 5000);
MGDO. DrawDisplayObject (do1, 5);
MGIW. SetPos(1000, 1000);
MGDO. DrawDisplayObject (do1, 26);
...
MGDO. LoadNamedDisplayObjects (do2, 'NAMES.DO'
'Bert, Carl');
MGDO. DrawDisplayObject (do2, MGDO. DisplayObjectId (do2, 'Carl'));
...
MGDO. DisposeDisplayObjects (do1);
MGDO. DisposeDisplayObjects (do2);
```

In this program sketch, we load display objects using both numbers and names. When drawing however, numbers are always used.

**Portability**

Normally, display object libraries are used, just like symbol libraries, in order to create reusable display objects. Display object libraries, however, give greater freedom than symbol libraries in choosing drawing commands, colors, etc. It is therefore important to use commands that make the objects as general and as portable as possible.

- Use world coordinates (**MGIW** or **MGW/MGW87**) in order to not limit the size to a particular device. Describe all display objects in one and the same standard system (like 0..10000 with **MGIW** or 0.0..1.0 with **MGW**). You can change the dimensions of display objects when drawing by simply changing the world coordinates in the playback window.
· Use integer based world coordinates which give more compact display object files and better execution speed.

· Use relative instead of absolute positions.

· Use `MG.MatchForeground` and `MG.MatchBackground` instead of device specific colors. This makes it easier to move display objects between different devices and between color and monochrome screens.

· Use the generic line styles and patterns in `MGPat`.

### Display Objects versus Procedures

Display objects are based on display lists that in turn consist of graphics commands from the `MultiGraphics` arsenal. Often display lists are created by recording drawing commands during execution of the program. You can also produce a display list by "manually" placing code and parameters for graphics commands in the list. The question is: When should you code the graphics instruction directly in the program source, and when should you place them in display object libraries?

Here are some advantages in using display object libraries:

· General libraries of display objects can be used by many different programs without coding.
· Display object libraries can be used with all the programming languages and compilers that `MultiGraphics` supports.
· New display objects can be added and old objects can be modified externally without the program code being changed or recompiled. All programs that use a particular object library will then automatically use the new versions of the display objects.
· Display objects can be created dynamically and interactively during the execution of a program (e.g., by an object oriented drawing program).
· Display objects that are no longer needed can be deleted from memory.

Here are some advantages of putting the code directly in the programs:

· Data can be transferred with greater flexibility when calling procedures directly, compared to when executing display objects.
· Faster execution of the program (but the difference is often minimal).
· Optimized linking - only the code for those procedures that are really called are linked into the program. With display objects also, unused routines are often linked since it is difficult to determine at compilation time exactly which graphics commands that will be called during the execution of the program.

Error Handling

The unit **MGDO** uses the **SuperVisor** technique for error handling. Errors are taken care of automatically by error procedures (as a default **SV.GlbEP** is called) or manually by checking **SV.Result** after calling procedures and functions in **MGDO**.

Examples

**GADO** creates a simple display object library, saves this on disk, loads the display objects again and draws them repeatedly.

Constants

<table>
<thead>
<tr>
<th>unitName</th>
<th>'MGDO'</th>
</tr>
</thead>
<tbody>
<tr>
<td>copyright</td>
<td>'DATABITEN'</td>
</tr>
</tbody>
</table>

| invalidDisplayObjects | 1525 |
| displayObjectsUsed    | 1526 |
| invalidDisplayObjectId| 1527 |
| usedDisplayObjectId   | 1528 |
| invalidDisplayObjectsFile| 1529 |
| invalidDisplayObjectNames| 1530 |

Specific error codes in **MGDO**.

| maxDisplayObjects | 128 |

Maximum number of display objects in a display object library.
nullDisplayObjects

Constant for specifying a non-existent display object library.

Data types

DisplayObjectNameStr = STRING[12];

String type for display object names (max. length is 12 characters).

DisplayObjectInfo = RECORD
  id    : Word;
  name  : DisplayObjectNameStr;
  dl    : MG.DisplayList;
  loaded: BOOLEAN;
END;

Display object information with the display object's number, name, appropriate display list and a flag that states whether the object has been loaded from a display object file (TRUE) or has been added by the program (FALSE). It is used as a parameter type in GetDisplayObjectInfo.

DisplayObjectSet = SET OF [0..maxDisplayObjects - 1];

Set of display object numbers. Used as a parameter type when calling LoadDisplayObjects.

DisplayObjects

Data type for a library of display objects.

Variables

revision : Word;
EP       : SV.ErrProc;

Revision number and local error handling procedure for the MGDO unit.
Procedures & Functions

Create Display Object Libraries

**CreateDisplayObjects**

```pascal
P CreateDisplayObjects(VAR dObjects: DisplayObjects (*O*));
```

Creates a new and empty library of display objects attached to the variable `dObjects`.

Display object libraries are implemented as lists, compatible with lists in the unit `DBList`.

**SV.Result**

- `MGErr.outOfMem` not enough memory
- `DBList.outOfSpace` not enough memory

```pascal
VAR
  icons: MGDO.DisplayObjects;
  ...
  MGDO.CreateDisplayObjects(icons);
  ...
  MGDO.SaveDisplayObjects(icons,'ICONS.DO');
  MGDO.DisposeDisplayObjects(icons);
```

**DisposeDisplayObjects**

```pascal
P DisposeDisplayObjects(VAR dObjects:DisplayObjects (*IO*));
```

Deletes a display object library, attached to the variable `dObjects`, from memory. The object's display lists are deleted only if the object has been loaded from disk.

**SV.Result**

- `invalidDisplayObjects` invalid library of display objects

Errors from `DBList` and `MGDL` are also reported.
Information about Display Objects

GetDisplayObjectInfo

```pascal
P GetDisplayObjectInfo(    dObjects: DisplayObjects;
    id      : Word;
    VAR info    : DisplayObjectInfo (*O*));
```

Returns information about a particular display object, with number `id`, in the display object library `dObjects`. The information, which is of the type `DisplayObjectInfo`, is placed in the variable `info`.

In `info`, there is information about the display object's name, display list and if the object has been loaded from disk or not. If the display object with stated number is not available, `info.id` is set to `65535`, `info.name` to the empty string and `info.dl` to `MG.nullDisplayList`.

**SV.Result**

- `invalidDisplayObjects` invalid library of display objects
- `invalidDisplayObjectId` the display object number must be `< 128`

Errors from `DBList` are also reported.

```pascal
VAR
    do    : MGDO.DisplayObjects;
    doInfo: MGDO.DisplayObjectInfo;
    dePtr : MG.DisplayElementPtr;
    ..
    MGDO.GetDisplayObjectInfo(do,5,doInfo);

(* Append to the display list in display object *)
MGDL.AppendDisplayElement(doInfo.dl,dePtr);
MGDO.DrawDisplayObject(do,5);
```

DisplayObjectId

```pascal
F DisplayObjectId(dObjects: DisplayObjects;
    name    : DisplayObjectNameStr): Word;
```

Returns the *object number* of a display object called `name` in the display object library `dObjects`. If an object with this name is not found in the display object library, `65535` is returned.
The function searches through the display object library, display object by display object, until it finds the object name. Since this takes time, you should avoid searching for the same object repeatedly - it is better to store the object number in a variable once and for all.

**SV.Result**
invalidDisplayObjects invalid library of display objects

Errors from **DBList** are also reported.

```pascal
VAR
do : MGDO.DisplayObjects;
doId: Word;
.
doId:= MGDO.DisplayObjectId(do,'Volvo850');
IF doId < MGDO.maxDisplayObjects THEN
  MGDO.DrawDisplayObject(do,doId);
.
```

### DisplayObjectName

```pascal
F DisplayObjectName(dObjects: DisplayObjects;
  id : Word): DisplayObjectNameStr;
```

Returns the *object name* of a display object with object number **id** in the display object library **dObjects**. If an object with this number is not found in the display object library, an empty string is returned.

**SV.Result**
invalidDisplayObjects invalid library of display objects
invalidDisplayObjectId the display object number must be < 128

Errors from **DBList** are also reported.

```pascal
VAR
do: MGDO.DisplayObjects;
.
WriteLn(MGDO.DisplayObjectName(do,10));
```
Add and Delete Display Objects

**AddDisplayObject**

```
P AddDisplayObject (dObjects: DisplayObjects;
    id      : Word;
    dl      : MG.DisplayList;
    name    : DisplayObjectNameStr);
```

Stores a new display object with object number `id` in the display object library `dObjects`. The display object is described in the display list `dl` and is given the name `name`.

There must not be a display object with the same number in the display object library. If you want to change a display object, this must first be deleted with `DeleteDisplayObject`. A display object library contains the maximum of `maxDisplayObjects` (128) display objects.

The object name can use all the characters in the computer's character set with the exception of the comma (',' is used to separate object names when loading from disk). Examples of permitted names are "DBLOGO", "Tortoise", "Long arrow", "123-calc", etc. The case of the letters is significant, i.e. MGDO differentiates between upper and lower case letters in object names. "DBLOGO", "dblogo", "DBlogo", etc, are regarded as different names. The maximum length of the name is 12 characters (sizeof(DisplayObjectNameStr)).

*AddDisplayObject* does not check that the object name is unique in the display object library - the responsibility for this is up to you!

The display object refers directly to the original display list, which is why this must not be deleted as long as the display object library is still in memory. Only after calling `DisposeDisplayObjects` or `SaveDisplayObjects` can the original display list be deleted (the display list's information is stored in the display object file).
SV.Result
MGErr.invalidDisplayList invalid display list
invalidDisplayObjects invalid library of display objects
invalidDisplayObjectId the display object number must be < 128
usedDisplayObjectId the display object number already used
invalidDisplayObjectNames object name must not contain a comma

Errors from DBList are also reported.

VAR
cicons: MGDO.DisplayObjects;
disk,
arrow: MG.DisplayList;
..
MGDO.AddDisplayObject (icons, 120, disk, 'DISKETTES');
MGDO.AddDisplayObject (icons, 100, arrow, 'ARROW');
MGDO.SaveDisplayObjects (icons, 'ICONS.DO');
..

DeleteDisplayObject

P DeleteDisplayObject (dObjects: DisplayObjects;
id : Word);

Deletes a display object with the object number id from the display object library dObjects. If the object number has not been used nothing is done. The object's display list is deleted only if the object has been loaded from disk.

SV.Result
invalidDisplayObjects invalid library of display objects
invalidDisplayObjectId the display object number must be < 128

Errors from DBList are also reported.
Draw display objects

**DrawDisplayObject**

```pascal
P DrawDisplayObject(dObjects: DisplayObjects;
  id      : Word);
```

Draws the display object **id** in the display object library **dObjects**, i.e. plays back the graphics commands in **id**'s display list. The drawing is done in the current window using the current window mode, which however can be changed by commands in the display object. Therefore, nothing can be known about the window mode after the call is made.

If you want to protect the window mode, it can be read before the playback by calling `MG.GetWinMode` and perhaps also `MGIW.GetWorldMode`, `MGW.GetWorldMode` and `MGIO.GetIOMode`. After the playback, the mode is reset with the corresponding mode commands, i.e. `MG.SetWinMode`, `MGIW.SetWorldMode`, `MGW.SetWorldMode` and `MGIO.SetIOMode` respectively.

If the display object with number **id** is not available, an error message is returned.

The current window must be enhanced with all capabilities that are used by graphics commands in the display object. Moreover, all graphics commands used must be linked (with `MG.Link`, `MGIW.Link`, etc. - see "MGDL - Display Lists" for more information)

**SV.Result**

- **invalidDisplayObjects**: invalid library of display objects
- **invalidDisplayObjectId**: display object with this number is not available

Errors from **MGDL** and **DBList** are also reported.

```pascal
VAR
  icons: MGDO.DisplayObjects;
  ..
  MG.Link; MGIW.Link; ..
  ..
  MG.EnhanceWin(MG.CurWin, MGIW.cap);
  ..
  MGIW.SetPos(7000,1000);
  MGDO.DrawDisplayObject(icons,
    MGDO.DisplayObjectId(icons, 'DISKETTE'));
```
Display Object Files

**SaveDisplayObjects**

```pascal
P SaveDisplayObjects(dObjects: DisplayObjects;
                      fname   : String);
```

Saves a display object library, `dObjects`, in a display object file with name and path in `fname`. All display objects in the library, including the object's display lists are stored. See however, the additional information about storing graphics commands with *reference parameters* in "MGDL - Display Lists".

The information in symbols, fonts and images that are included in display objects are never saved in the display object files. Instead, they must be kept available in ordinary symbols, font and image files when the display object is loaded again from a display object file.

Any possible file with the same name is written over.

**SV.Result**

Errors from `DBList` are reported.

**LoadDisplayObjects**

```pascal
P LoadDisplayObjects(VAR dObjects : DisplayObjects; (*O*)
                      fname    : String;
                      doSet    : DisplayObjectSet);
```

Creates a new display object library, `dObjects`, and loads one or more display objects from the display object file, with name and path in `fname`, to this library. The object numbers of the desired display objects are specified with the number set `doSet`.

**SV.Result**

- `MGErr.outOfMem` not enough memory
- `displayObjectsUsed` the library `dObjects` already in use

Errors from `MGDL` and `DBList` are also reported.
VAR
  icons: MGDO.DisplayObjects;
...
MGDO.LoadDisplayObjects(icons,[0,10..15]);
MGDO.DrawDisplayObject(icons,11);

**LoadNamedDisplayObjects**

```pascal
PROCEDURE LoadNamedDisplayObjects(VAR dObjects: DisplayObjects;
  fname   : String;
  doNames : String);
```

Creates a new display object library, **dObjects**, and then loads one or more display objects from the display object file, with name and path in **fname**, to this library. The object names for the desired display objects are specified in the text string **doNames** using a comma between each object name.

**MGDO** differentiates between upper and lower case letters in the object name. "DBLOGO", "dblogo", "DBlogo", etc., are interpreted as different names.

**SV.Result**

MGErr.outOfMem not enough memory

displayObjectsUsed the library **dObjects** already in use

Errors from **MGDL** and **DBList** are also reported.

VAR
  icons: MGDO.DisplayObjects;
...
MGDO.LoadNamedDisplayObjects(icons, 'DISKETTE,ARROW');
MGDO.DrawDisplayObject(icons,
  MGDO.DisplayObjectId(icons, 'ARROW'));

MultiGraphics & Borland Pascal/Turbo Pascal  DATABITEN
The unit MGOOut contains procedures for the printing of graphics via serial and parallel ports as well as for saving printouts on disk. The procedures in MGOOut are used with graphics devices such as dot matrix printers, laser printers, ink jet printers, pen plotters, etc. When printing via serial ports, it is possible to set the communication speed, interface parameters etc.

Files on disk are used when you want to save graphics for printing later, or when the printing is to be done on another computer with access to a graphics device, or if you want to debug the printouts for errors. The output file's name and directory can be chosen freely.

Two graphics devices cannot simultaneously use the same printing routine (e.g., COM1 or DISK)!
How to Use Printout Procedures

A printout procedure is installed by calling \texttt{MG.SetDeviceOutput}. The call must be made before the graphics device is activated with \texttt{MG.SetDevice}. For example:

```pascal
VAR
  driver1, driver2, driver3: MG.Device;
.. MG.LoadDevice(driver1,'EPS.MGA');
MG.SetDeviceOutput(driver1,MGOut.LPT1);
MG.SetDevice(driver1);
.. MG.LoadDevice(driver2,'HPGL.MGA');
MGOut.InitCOM2(MGOut.hardware,
  9600,8,1,
  MGOut.noParity);
MG.SetDeviceOutput(driver2,MGOut.COM2);
MG.SetDevice(driver2);
.. MG.LoadDevice(driver3,'HPL.MGA');
MGOut.InitDISK('laser.prn');
MG.SetDeviceOutput(driver3,MGOut.DISK);
MG.SetDevice(driver3);
.. MGOut.CloseDISK;
```

As an alternative, you may use the enumeration type \texttt{DeviceOutput} with the initiated array \texttt{OutProc}. We can then rewrite the example above as:

```pascal
VAR
  driver1, driver2, driver3: MG.Device;
.. MG.LoadDevice(driver1,'EPS.MGA');
MG.SetDeviceOutput(driver1,MGOut.OutProc[MGOut.outLPT1]);
MG.SetDevice(driver1);
.. MG.LoadDevice(driver2,'HPGL.MGA');
MGOut.InitCOM2(MGOut.hardware,
  9600,8,1,
  MGOut.noParity);
MG.SetDeviceOutput(driver2,MGOut.OutProc[MGOut.outCOM2]);
MG.SetDevice(driver2);
.. MG.LoadDevice(driver3,'HPL.MGA');
MGOut.InitDISK('laser.prn');
MG.SetDeviceOutput(driver3,MGOut.OutProc[MGOut.outDISK]);
MG.SetDevice(driver3);
.. MGOut.CloseDISK;
```
The main benefit with the last approach is that the standard printout procedures may be referenced by a simple enumeration type instead of procedure variables.

When printing to a file on disk, it is important that the file is closed *after* the printing driver is closed but *before* the program is terminated. Otherwise data can be lost!

**Inspecting, Modifying Data and Messages**

**MGOut** also makes it convenient to install peeking procedures to supervise printing and to bring messages to the user during printing. This is done by calling **SetPeekProc**.

At other times, it may be necessary to modify data before it is sent to the printer. Some printers are not fully compatible with standards, so printer codes have to be changed, or you may want to avoid sending certain initializing or resetting codes. This is easily done by building your own printout procedure which accomplishes the desired actions before the printout procedures in **MGOut** are called. For example:

```pascal
PROCEDURE MyLPT1(    dataAdr: Pointer;
                      count  : Word;
                      VAR error  : INTEGER);
BEGIN
  (* Inspect & modify data in dataAdr^ and count *)
  ..
      LPT1(dataAdr,count,error);
  END;
  ..
MG.SetDeviceOutput(printer,MyLPT1);
  ..
```
Constants

unitName 'MGOut'
copyright 'DATABITEN'
invalidCOMData 1425
initCOMError 1426
diskInUse 1427
diskNotInUse 1428
invalidDeviceOutput 1429

Special error codes for the unit MGOut.

Data Types

DeviceOutput = (outLPT1, outLPT2, outLPT3, outCOM1, outCOM2, outDISK); Enumeration type for standard output procedures.

FlowControl = (noControl, hardware, xonxoff);

Data types that define protocols when printing to serial devices.

noControl means no checking is carried out with the risk that data is transmitted faster than what the receiving graphics device is able to handle. Using noControl, it is sufficient to use a one-way connection from the computer to the device.

hardware means that the flow of data is controlled by a special cable from the device back to the computer. Normally, the computer's CTS pin (Clear-To-Send) is connected to the device's RTS pin (Request-To-Send).

xonxoff means that the device controls the flow of data from the computer via a program by sending a particular start code, normally ^S (ASCII 19), and stop code, normally ^Q (ASCII 17).

Parity = (noParity, oddParity, evenParity);
By parity is meant a method for checking that data is transferred correctly. Parity checking is seldom used when printing on normal graphics devices such as printers and plotters. noParity is therefore the default value.

Variables

EP: SV.ErrProc;
This gives the current error procedure in the unit MGOut.

OutProc = ARRAY [DeviceOutput] OF MG.DeviceOutputProc;

This array is initialized to contain references to the printout procedures LPT1, LPT2,...DISK. Thus OutputProc[outCOM1] contains a reference to the procedure COM1. The array elements are mostly used when calling MG.SetDeviceOutput.

MG.SetDeviceOutput(driver,MGOut.OutProc[MGOut.outCOM1]);

The above code has same effect as:

MG.SetDeviceOutput(driver,MGOut.COM1);

Procedures

Peek Procedures

SetPeekProc

P SetPeekProc(outDev : DeviceOutput;
              PeekProc: MG.DeviceOutputProc);

This procedure installs the inspection procedure PeekProc, of MG.DeviceOutputProc type, in the output routine named in outDev (outLPT1, outLPT2,...outDISK). This peek procedure is called on every
call to the printing procedure named in `outDev` before any data has been sent to the printer or disk.

Peek procedures can be used to supervise the transmission of data to a printing device or disk file and also to display appropriate messages on screen during a lengthy printer session. The peek procedure should set its `error` parameter to 0 if data is accepted for printing, otherwise to a value <> 0. This error code is then brought back to the calling `Copy` procedure.

```pascal
VAR
  clock0: LongInt;

{$F+}
PROCEDURE UserMessage(    data : Pointer;
                           count: Word;
                           VAR error: INTEGER);

CONST
  interval = MGClock.ticksPerSecond;
VAR
  clock: LongInt;
  cwin : MG.Window;
BEGIN
  clock:= MGClock.ClockNow;
  IF (clock > clock0 + interval) OR
     (clock < clock0) THEN BEGIN
    cwin:= MG.CurWin;
    MG.SetWin(msgWin);
    MG.SetDrawMode(MG.complement);
    MG.DrawStr('Printing...');
    MG.SetWin(cwin);
    clock0:= clock;
  END;
  error:= 0;
END; (*UserMessage*)
...
clock0:= 0;
MGOut.SetPeekProc(MGOut.outCOM1,UserMessage);
MG.SetDeviceOutput(pr,MGOut.COM1);
...
```
Parallel Ports

**LPT1, LPT2, LPT3**

\[
P \text{LPT1}(\text{dataAdr: Pointer;}\text{, count : Word;}\text{, VAR error : INTEGER});
\]

\[
P \text{LPT2}(\text{dataAdr: Pointer;}\text{, count : Word;}\text{, VAR error : INTEGER});
\]

\[
P \text{LPT3}(\text{dataAdr: Pointer;}\text{, count : Word;}\text{, VAR error : INTEGER});
\]

These are procedures for printing via parallel ports 1 (LPT1), 2 (LPT2) and 3 (LPT3) respectively. They are used as procedure parameters for `MG.SetDeviceOutput`. On the other hand, they are seldom called directly, but if you want to transfer some information or setup data to a printer, you can call these procedures directly.

```pascal
VAR
  printer: MG.Device;
  ..
MG.LoadDevice(printer, 'EPS.MGA');
MG.SetDeviceOutput(printer, MGOout.LPT1);
MG.SetDevice(printer);
  ..
```
Serial Ports

**COM1, COM2**

```pascal
P COM1(    dataAdr: Pointer;
          count  : Word;
          VAR error  : INTEGER);

P COM2(    dataAdr: Pointer;
          count  : Word;
          VAR error  : INTEGER);
```

These are procedures for printing via serial ports 1 (COM1) and 2 (COM2) respectively. They are used as procedure parameters for `MG.SetDeviceOutput`. On the other hand, they are seldom called directly, but if you want to transfer some information or setup data to a printer, you can call these procedures directly.

Before printing is started the serial port must be initialized using the desired communication parameters. This is done from DOS (MODE command) or by calling `InitCOM1` or `InitCOM2`.

```pascal
VAR
  plotter: MG.Device;
  ...
MG.LoadDevice(plotter,'HPGL.MGA');
MGOut.InitCOM2(MGOut.hardware,9600,8,1,MGOut.noParity);
MG.SetDeviceOutput(plotter,MGOut.COM2);
MG.SetDevice(plotter);
  ...
```

**InitCOM1, InitCOM2**

```pascal
P InitCOM1(flow: FlowControl;
          baud: LongInt;
          bits: Word;
          stop: Word;
          par : Parity);
```
P InitCOM2(flow: FlowControl;
  baud: LongInt;
  bits: Word;
  stop: Word;
  par : Parity);

These procedures initialize serial port 1 and serial port 2 respectively. flow specifies the choice of flow control (normally hardware), baud specifies the baud rate (300, 1200, 2400, 4800 or 9600 baud, normally 9600), bits decides the number of data bits (7 or 8, normally 8), stop the number stop bits (1 or 2, normally 1) and finally par, the parity (normally noParity).

If InitCOM1 or InitCOM2 is not called before printing, the default values or those values that have been given with the MODE command in DOS are used.

SV.Result
initCOMError serial port is missing or data error
invalidCOMData invalid parameters

MGOOut.InitCOM1(MGOOut.hardware, 9600,8,1,MGOOut.noParity);
IF SV.Result <> SV.ok THEN ..

Files on Disk

DISK

P DISK(dataAdr: Pointer;
  count : Word;
  VAR error : INTEGER);

This is the procedure for printing files on disk. The printout file is specified by calling InitDISK. DISK is used as a procedure parameter for MG.SetDeviceOutput but is seldom called directly.

Before DISK is used, a file must be opened using InitDISK. Printing must be terminated with CloseDISK.
VAR
  plotter: MG.Device;
 ..
MG.LoadDevice(plotter,'HPGL.MGA');
MGOout.InitDISK('C:\output\hpgl.prn';
MG.SetDeviceOutput(plotter,MGOout.DISK);
MG.SetDevice(plotter);
 ..
MGOout.CloseDISK;

---

**InitDISK**

```pascal
P InitDISK(fileName: String);
```

This procedure opens the file on disk to which the printing via **DISK** (see above) are directed. **fileName** is a string that, besides the filename, can contain the name of the desired disk and path. In order to guarantee that all data is printed, the file must be closed when the work is terminated (**CloseDISK**) This is also done automatically by the termination routine in **MGOout**.

**SV.Result**

- **diskInUse**  
  - `< 0` DOS error in attempting to create and open the file

```pascal
MGOout.InitDISK('C:\output\temp.prn');
```

---

**CloseDISK**

```pascal
P CloseDISK;
```

This procedure closes a previously opened file on disk. **CloseDISK** must be called after the termination of printing, otherwise you cannot be sure that all data was actually written to the file.
By *event handling* is meant a technique for dealing with data from input devices such as keyboards and mice, as well as the system clock, disks, serial ports etc. These input devices produce a stream of events such as keys being pressed, mouse movements, the pressing, double clicking and releasing of mouse buttons, ticks from the system clock, etc.

From *MultiGraphics'*s point of view, an *event* is a "data package" which is delivered at

- a particular *point in time* and
- from a particular *input device*. 

Fig 1: Common event generators
An event generator is an input device which creates events. The standard event generators in MultiGraphics are the keyboard, the mouse and the clock. Event generators are numbered from 0 to 15.

An event type is a subset of those events that are produced by a particular event generator. Examples of event types are:

- **keyboard**
  - a key being pressed

- **mouse**
  - a movement of the mouse
  - pressing the left mouse button
  - releasing the right mouse button
  - double clicking the left mouse button

- **clock**
  - tick
  - single alarm
  - repeating alarm

An event is characterized by

- the point in time when the event occurred,
- the event generator responsible for the event,
- a set of event types,
- other unique information for the event generator and event types.

The basic type for events is the data type Event which is defined in MGEv.

For example, a mouse event contains information about the point in time for the event, that the event generator is the mouse, the set of mouse event types that have occurred (e.g., the mouse movement and the pressing of the left button), as well as unique information concerning the mouse generator such as the mouse's position and which mouse buttons that are held down.

Event handling in MultiGraphics is very powerful and flexible. It has no specific requirements of how data from various event generators is to be formatted - except for the minimum requirements of information about the point in time, the event generator and the event types. Other information may be tailored to suit the event generator.

### The Event Queue

The event handling in MGEv sees to it that all events are placed in order in one common queue, the event queue. Events are queued in strict order of time with the first event in the queue being dealt with first. Expressed in
computer terminology, we say that the event queue is of the FIFO type (First In, First Out).

The program's task is quite simply to deal with the event queue, event by event, and to implement the instructions given by the data in each respective event.

An event controlled program is normally characterized, therefore, by a program loop which fetches an event from the event queue, processes that event, fetches the next event, and so on.

```
loop
    read event;
    process the event;
end;
```

Since the event queue may sometimes be empty, or not have a appropriate event, the loop is often supplemented as follows:

```
loop
    if event has occurred
        read event;
        process the event;
    else
        do something in the meantime;
    end
end;
```

In comparison, the unit MGIO defines a large number of routines for keyboard handling. Common to all of these is that they are polling routines, which means that the program itself must take the initiative in requesting the information from this or that input.
device (i.e. the keyboard for MGIO). This technique is sufficient for supervising one individual input device, but becomes problematic if several input devices deliver information simultaneously. How often and in which order should the input devices be read? What happens if one input device does not produce data when we expect it to? And so on, and so on. In cases like this, event handling often offers a simpler and better solution.

Events in the event queue may be inspected using EventsPending, CheckNextEvent and NextEventGen as well as be read with RdNextEvent and RdMatchingEvent.

The maximum number of events that can be held in the queue at any one time is limited (default is 128, see below). If the program cannot process events at the rate they are produced, the queue will eventually become full. When the queue is full, the latest event received is placed last in the queue. The previous event in this position is deleted from the queue and its information is lost!

Event Generators

Event handling in MGEv can cope with up to 16 event generators simultaneously. Event generators are given a number within the range of 0..15. Of these numbers, the following are reserved:

<table>
<thead>
<tr>
<th>Generator</th>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nullGen</td>
<td>0</td>
<td>&quot;empty&quot; event generator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- does nothing</td>
</tr>
<tr>
<td>clockGen</td>
<td>1</td>
<td>the system clock</td>
</tr>
<tr>
<td>kbdGen</td>
<td>2</td>
<td>the keyboard</td>
</tr>
<tr>
<td>mouseGen</td>
<td>3</td>
<td>the mouse</td>
</tr>
</tbody>
</table>

We recommend that you restrict your own generators to the interval 10..15, as future versions of MultiGraphics may need more standard generators.

Reporting Events

The event handling is started automatically when the unit MGEv is initialized. It can be stopped by using StopEventHandler and restarted with StartEventHandler.

Under DOS, all event generators report their events to the event handler in MGEv.
using the interrupt technique. When the unit \texttt{MGEv} is initialized, the event handler installs an interrupt handler linked to a particular interrupt vector, \textit{the event interrupt vector}. An event generator can find out which vector is being used by calling the function \texttt{CurEventIntrVec}.

When initialized \texttt{MGEv} installs the interrupt vector specified by the variable \texttt{PasSys.eventIntrVec} (default is \texttt{F1H}). You can restart the event handling using \texttt{StartEventHandler} and then choose another interrupt vector.

Data about the event, such as which event generator has been used, which event types etc., are transferred when the event interrupt occurs via the 80x86 processor's register. The \texttt{AX} register always contains the number of the event generator, while the other registers can be used freely. The \texttt{BX} register is normally used to report the \textit{event types} that have occurred.

\section*{Event Service Routines}

The event handler in \texttt{MGEv} receives events and put them in the last position in the event queue. Each event is time stamped (the field \texttt{ticks} in the event structure \texttt{Event}) and earmarked with the number of the event generator (the field \texttt{gen} in the event structure). After that, the \textit{event service routine} of the current event generator is called. This is a function of the type:

\begin{verbatim}
EventServerProc =
  FUNCTION(VAR data: MGEv.EventData;(*I*)
            VAR ev  : MGEv.Event    (*IO*) ): BOOLEAN;
\end{verbatim}

\texttt{data} (see \texttt{EventData} type below) is a record which under DOS contains a time stamp for the message and the values of the microprocessor registers \texttt{AX}, \texttt{BX}, \texttt{CX} and \texttt{DX} as set by the event generator. \texttt{ev} contains the current event (of the type \texttt{Event}) in the event queue.

An event service routine adds relevant information to the \texttt{ev} event in the queue using data from the variable \texttt{data}. It carries out other tasks specific for the event generator (e.g., draws a mouse pointer), and returns \texttt{TRUE} if everything has gone well. Clock time in \texttt{ev.ticks} as well as the event generator of the event in \texttt{ev.gen} are already assigned before the event server procedure is called.

Event service routines are installed by \texttt{InstallEventServer}.

\section*{Inside the Event Queue}

The event queue is implemented as an array of the element type \texttt{Event}. The maximum number of events in the queue is set, when \texttt{MGEv} is initialized, to the default value
Ref 27-6

MGEv - Events

128. This value can be changed by restarting the event handling again using `StartEventHandler`.

Events in the event queue may be inspected (`EventsPending`, `CheckNextEvent` and `NextEventGen`) and be read (`RdNextEvent` and `RdMatchingEvent`).

Error Handling

Take note that several of the routines in MGEv (`CurEventIntrVec`, `CurMaxEvents`, `FlushEvents`, `EventsPending`, `CheckNextEvent`, `NextEventGen`, `RdNextEvent` and `RdMatchingEvent`), for speed reasons, do not use the error handling in SV. The result of these calls can not be checked using `SV.Result`.

Program Examples

**GAEvents** shows basic event handling using clock events (`MGClock`), keyboard events (`MGKbd`) and mouse events (`MGMouse`). The program also shows mouse emulation using the arrow keys of the keyboard.

**GAMouse** shows mouse event handling and the use of different types of pointer symbols (complement symbol, stamp symbol, mask symbol, stamp image), using the mouse buttons, double clicking, mouse event procedures to correlate the look of the mouse pointer to the mouse's position, as well as how to protect graphics and mouse pointers.

**GAClock2** shows the concurrent handling of an analog clock, the keyboard and the mouse with the help of event handling.

**Creator** is an animated game showing animation techniques and advanced event handling (see "Animation on the PC").
Constants

unitName 'MGEv'
copyright 'DATABITEN'

eventHandlerActive 1150
eventHandlerNotActive 1151
invalidServerProc 1152
invalidEventGenerator 1153

These are error codes for specific errors in the unit MGEv.

stopEvents 255

maxEventGen 15

The maximum permitted event generator number in MultiGraphics.

nullGen 0
clockGen 1
kbdGen 2
mouseGen 3

The numbering of the standard event generators that are included in MultiGraphics. nullGen denotes the "non-existent" event generator.

allEventGens [0..15]

This is a predeclared set constant which includes all event generators. It can be used, for example, in calling RdMatchingEvent when you want to read events from all event generators.

allEvents [0..15]

This is a predeclared set constant which includes all event types. It may be used, for example, in calling RdMatchingEvent when you want to read all event types.
Data Types

```
EventGen = 0..Word8(maxEventGen);
```

Data type for numbering of event generators. Numbers 0..3 are reserved for standard generators (see "Constants" above), while 4..15 can be used for the your own event generators, however, we recommend that you restrict yourself to then interval 10..15.

```
EventGenSet = SET OF EventGen;
```

Set type for event generators. It is used as a parameter type in RdMatchingEvent to specify which event generators that are to be read. Using allEventGens you can read all input devices.

```
Event = RECORD
ticks : Integer32;
gen : EventGen;
CASE INTEGER OF
  0: (events : BitSet16;
     data : ARRAY[0..10] OF Byte);
  1: (logEvents : Word16;
     logData : ARRAY[0..10] OF Byte);
END;
```

The generic type for events and that is independent of any specific event generator. The first three fields in Event are the same for all event data types, while the last data/logData field can be implemented in different ways depending on the needs of a specific event generator. The third field is declared as a Pascal variant field events:BitSet16 (SET OF 0..15) and logEvents:Word16 respectively. The set variant is used in set expressions (elegant) and the Word16 variant in logical arithmetic expressions (faster in Turbo Pascal versions before 7.0). An event generator need not use all bytes in the field data/logData, but neither should it hold more data than the size of this field (see MGClock.ClockEvent, MGKbd.KbdEvent and MGMouse.MouseEvent for concrete event data types).

```
EventMatch = (matchAnyEvent,
              matchAllEvents,
              matchEventsExactly);
```
Data type to control the type of matching between event types in the parameter `evMask` and the event types that have actually occurred (the field `Event.events`) when calling `RdMatchingEvent`.

`matchAnyEvent` is satisfied if any of the event types in `evMask` take place (`evMask*events <> the empty set`).

`matchAllEvents` requires that all event types in `evMask` occur. However, nothing prevents other event types from also taking place (`evMask - events = the empty set`).

`matchEventsExactly` requires exact agreement between `evMask` and the events occurred (`events = evMask`).

```pascal
EventData = RECORD
  ticks : Integer32;
  CASE INTEGER OF
    1: (DX, CX, BX, AX : Word);
    2: (DL, DH, CL, CH, BL, BH, AL, AH : Byte);
    3: (DX, CX, BX, AX : PasSys.QuadWord);
  END;
END;
```

Data structure used internally to bring event information from the event routines in `MGEv` to the event server functions (located in `MGClock`, `MGKbd`, `MGMouse`, etc). The `ticks` field is stamped with the current clock time at the arrival of the event, while `DX`, `CX`, `BX` and `AX` (and their variants) hold register values as set by the event generator before calling `MGEv`. `AX` always contains the number of the event generator. The other registers are at your (the event generator's) disposal.

```pascal
EventServerProc =
  FUNCTION(VAR data: MGEv.EventData; (*I*)
           VAR ev : MGEv.Event (*IO*)): BOOLEAN;
```

This is the procedure type for event service functions. Each event generator must install its own function of the type `EventServerProc`.

When a call is made, the parameter `data` contains a variable of `EventData` type (see above). From this, the service routine retrieves information about the event's type as well as other specific information for the event.

The variable parameter `ev` contains the current event in the event queue which is to be updated by the service routine (see "Event Service Routines" above).
The service function will return TRUE if the call was successful, otherwise FALSE.

Variables

revision: Word;
EP      : SV.ErrProc;

Procedures & Functions

Event Service Routines

**InstallEventServer**

```
P InstallEventServer(Server: EventServerProc;
                      gen   : EventGen);
```

This procedure installs the event service function `Server` for the event generator `gen (1..15)`. From then on, the event handler in MGEv calls this function after each event generated by `gen`. The service routine must be of the type `EventServerProc` (this is described under "Data Types" above). `InstallEventServer` must be called before the event generator is able to use these services in MGEv.

The event generators for time (MGClock) and keyboard (MGKbd) automatically call `InstallEventServer` from their initializing procedures.

SV.Result
invalidServerProc     invalid service function (NIL)
invalidEventGenerator only the range 1..15 is allowed
RemoveEventServer

P RemoveEventServer(gen: EventGen);

This procedure removes a previously installed service routine for the event generator gen (1..15). New events that are generated by gen are ignored from now on.

SV.Result
invalidEventGenerator only the range 1..15 is allowed

Starting the Event Handling

StartEventHandler

P StartEventHandler(intrVec : Word;
            maxEvents: Word);

This procedure allocates (conventional) memory for the event queue and installs a service routine for the event interrupt vector intrVec, as well as initializes the event handling. The maximum number of elements in the event queue is specified by maxEvents. The event vector intrVec is used to "send" events as interrupts to the event handler in MGEv.

Any previously installed service functions for events (InstallEventServer) are uninstalled. N.B. that no service routine is installed automatically - this must be done by calling InstallEventServer.

When MGEv is initialized, StartEventHandler is called automatically using the default interrupt vector intrVec = F1H and maxEvents = 128. Under normal conditions, you need not call StartEventHandler from your own program.

If you want to change the event interrupt vector you may only use unused vectors. IBM recommend F1H..FFH (86H..F0H are used only by BASIC and therefore should be able to be used as well without risk).

It is important that StartEventHandler is called before events are sent via the event interrupt vector. In all other cases, the program may crash. If you want to install one of your own event generators, we recommend that this generator, to be on the safe side, makes a call to MGEv.Init before the first
event interrupt occurs (this is done automatically by `MGClock`, `MGKbd` and `MGMouse`).

<table>
<thead>
<tr>
<th>SV.Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eventHandlerActive</code></td>
<td>the event handler is already active</td>
</tr>
<tr>
<td><code>outOfMem</code></td>
<td>not enough memory for the event queue</td>
</tr>
</tbody>
</table>

`StopEventHandler`:

```pascal
P StopEventHandler;
```

This procedure stops event handling in `MGEv`. The interrupt vector which was installed previously by `StartEventHandler`, or when the unit was initialized, is reset to its previous address. The event queue is deallocated. The installed service routines for the events are uninstalled.

It is important that the generating of events is stopped (i.e. calling the event handler via the installed event interrupt) before `StopEventHandler` is called. Otherwise the calls will be sent to an random memory address which, at best, will ignore the event interrupts and in the worst case will crash the program.

`StopEventHandler` is called automatically by the termination routine in `MGEv`, which means this need not be done manually at the end of the program.

<table>
<thead>
<tr>
<th>SV.Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eventHandlerNotActive</code></td>
<td>the event handler is not active</td>
</tr>
</tbody>
</table>

Data for the Event Handler

`CurEventIntrVec`:

```pascal
F CurEventIntrVec: Word;
```

This function returns the current event interrupt vector (installed by `StartEventHandler`). When event generators are installed, it is best to call `CurEventIntrVec` in order to find out which event vector is to be used for the event interrupt.
CurMaxEvents

F CurMaxEvents: Word;

This function returns the maximum number of events contained in the event queue (allocated by StartEventHandler).

Inspecting the Event Queue

EventsPending

F EventsPending: BOOLEAN;

This function returns FALSE if the event queue is empty, otherwise TRUE. It is sometimes used to check if there are events to be read (using RdNextEvent) in the queue.

EventsPending does not set the result code in SV and the result of the call can not be tested by invoking SV.Result.

IF MGEv.EventsPending THEN
  MGEv.RdMatchingEvent(...)

CheckNextEvent

P CheckNextEvent(VAR ev: Event (*O*));

This procedure inspects the first event in the event queue and returns the contents in the variable ev (of the generic event type). The event still remains in the event queue after the call is made.

CheckNextEvent does not set the result code in SV and the result of the call cannot be tested by invoking SV.Result.
VAR
  ev: MGEv.Event;
.. 
MGEv.CheckNextEvent(ev);
IF ev.gen = MGEv.kbdGen THEN MGEv.RdNextEvent(...)

**NextEventGen**

**F NextEventGen: EventGen;**

This function returns the number of the event generator that has generated the first event in the queue. The result nullGen means that the event queue is empty. The function is used to check if there are events in the queue, and if so which type, before you read the event using RdNextEvent.

**NextEventGen** does not set the result code in SV and the result of the call can not be tested by invoking SV.Result.

VAR
  clockEv: MGClock.ClockEvent;
  kbdEv  : MGKbd.KbdEvent;
  evAge  : LongInt;

CASE MGEv.NextEventGen OF
  MGEv.nullGen: ..(* Perform other tasks *)
  MGEv.clockGen:
    BEGIN
      MGEv.RdNextEvent(clockEv,evAge);
      .. (* Process clock event *)
    END;
  MGEv.kbdGen:
    BEGIN
      MGEv.RdNextEvent(kbdEv,evAge);
      .. (* Process keyboard event *)
    END;
END; (*case*)
Read Events

\textbf{RdNextEvent}

\begin{verbatim}
P RdNextEvent (VAR ev; (*O*)
    VAR age: Integer32 (*O*));
\end{verbatim}

This is the most important procedure for reading and processing events in the event queue. The first event in the queue is returned in the variable parameter \texttt{ev}, whose type is selected to match the event generator generating the event to be read. Variables of the generic type \texttt{Event} can always be used, but it is often better to use a type adapted to the particular event generator (\texttt{MGClock.ClockEvent}, \texttt{MGKbd.KbdEvent}, \texttt{MGMouse.MouseEvent}, etc.).

\texttt{ev} must be sized to accommodate the event read. Otherwise code or data will be destroyed with the most unpleasant consequences. To be safe you are recommended to size \texttt{ev} so that

\[ \text{SizeOf}(\texttt{ev}) \geq \text{SizeOf}(\texttt{MGEv.Event}) \]

If the event queue is empty, \texttt{RdNextEvent} waits until a new event takes place. To avoid blocking the program execution while waiting for events, it is often best to check \texttt{EventsPending} or \texttt{NextEventGen} before calling \texttt{RdNextEvent}.

The event read is removed from the queue.

The variable parameter \texttt{age} contains the "age" of the event, i.e. the time, expressed in clock ticks, between the event was generated and the time the event was read with \texttt{RdNextEvent}. Event based programs often prefer to ignore events that exceed a specific age (for example "old" mouse movements).

\texttt{RdNextEvent} does not set the result code in \texttt{SV} and the result of the call can not be tested by invoking \texttt{SV.Result}. If the event handler has not been initialized (\texttt{StartEventHandler}), there is a risk that the program will be dead-locked waiting for events to occur.
VAR
  clockEv: MGClock.ClockEvent;
  kbdEv : MGKbd.KbdEvent;
  mouseEv: MGMouse.MouseEvent;
  evAge : LongInt;
..
CASE MGEv.NextEventGen OF
  MGEv.clockGen:
    BEGIN
      MGEv.RdNextEvent(clockEv,evAge);
      IF (evAge < 10) AND
      (MGClock.repeatedAlarm IN clockEv.events) THEN
        ..
  MGEv.kbdGen:
    BEGIN
      MGEv.RdNextEvent(kbdEv,evAge);
      IF    kbdEv.key.wrd = MGIO.backspace.wrd THEN ..
      ELSE IF kbdEv.key.wrd = MGIO.tab.wrd THEN ..
      ELSE IF..
  MGEv.mouseGen:
    BEGIN
      MGEv.RdNextEvent(mouseEv,evAge);
      IF (evAge < 3) AND
      (MGMouse.mouseMoved IN mouseEv.events) THEN
        MG.DrawLineTo(mouseEv.scrX - MG.CurWinX,
                       mouseEv.scrY - MG.CurWinY);
      The same expression using logical arithmetic:
      IF (evAge < 3) AND
      (MGMouse.logMouseMoved AND
       mouseEv.logEvents <> 0) THEN
        MG.DrawLineTo(mouseEv.scrX - MG.CurWinX,
                       mouseEv.scrY - MG.CurWinY);
RdMatchingEvent

PROCEDURE RdMatchingEvent( genMask: EventGenSet;
                          evMask : BitSet;
                          match : EventMatch;
                          maxAge : Integer32;
                          VAR ev; (*O*)));

MultiGraphics & Borland Pascal/Turbo Pascal    DATABITEN
This is an enhanced procedure for reading and processing events in the event queue. The input parameters are:

- in `genMask`, the set of event generators that are to be supervised,

- in `evMask`, the set of desired event types. The interpretation of the individual bits depends on which event generator has produced the event. By setting all the bits (see the constant `allEvents`), all event types will be read,

- in `match`, the degree of agreement (`matchAnyEvent`, `matchAllEvents` or `matchEventsExactly`) between the desired event types in `evMask` and the event types that have occurred (see "Data types" above),

- in `maxAge`, the maximum acceptable "age" of the event to be read. The value is given in `timer ticks`.

The output parameter is the event read, `ev`, whose current type depends on which event generator has generated the event. Variables of the generic type `Event` can always be used, but it is often better to use a type adapted to the particular event generator (`MGClock.ClockEvent`, `MGKbd.KbdEvent`, `MGMouse.MouseEvent`, etc.).

`RdMatchingEvent` waits until an event takes place that fulfils the conditions with regard to the desired event generators, event types, matching and the event's age. Events that do not correspond to these conditions are ignored and removed (N.B.) from the event queue.

To avoid blocking the program execution while waiting for events, it is often best to check `EventsPending` or `NextEventGen` before calling `RdMatchingEvent`.

`RdMatchingEvent` does not set the result code in `SV` and the result of the call cannot be tested by invoking `SV.Result`. If the event handler has not been initialized (`StartEventHandler`), there is a risk that the program will be dead-locked waiting for events to occur.

```pascal
VAR
  clockEv: MGClock.ClockEvent;
  kbdEv  : MGKbd.KbdEvent;
  mouseEv: MGMouse.MouseEvent;
```
MGEv.RdMatchingEvent([MGEv.clockGen],
MGEv.allEvents,
MGEv.matchAnyEvent,
maxLongInt,
SizeOf(clockEv),
clockEv);
IF MGClock.repeatedAlarm IN clockEv.events THEN ..

MGEv.RdMatchingEvent([MGEv.kbdGen],
MGEv.allEvents,
MGEv.matchAnyEvent,
maxLongInt,
SizeOf(kbdEv),
kbdEv);
IF kbdEv.key.wrd = MGIO.backspace.wrd THEN ..
ELSE IF kbdEv.key.wrd = MGIO.tab.wrd THEN ..
ELSE IF ..

MGEv.RdMatchingEvent([MGEv.mouseGen],
[MGMouse.mouseMoved,
MGMouse.leftButtonPressed],
MGMouse.matchAllEvents,
Round(0.5*MG.ticksPerSecond),
SizeOf(mouseEv),
mouseEv);

MG.DrawLineTo(mouseEv.scrX - MG.CurWinX,
mouseEv.scrY - MG.CurWinY);

Flushing the Event Queue

FlushEvents

P FlushEvents;

This procedure empties the event queue of all events. It is used when you want to ignore previous events, or want to catch up with the events being generated, and so on.

FlushEvents does not set the result code in SV and the result of the call can not be tested by invoking SV.Result.
The unit **MGClock** contains the routines to read the system clock, convert the system clock's time to hours, minutes and seconds as well as generate clock events that are dealt with by the event handler in **MGEv**. **MGClock** also has complete routines for presenting digital and analog clocks.

Most computers have a clock circuit that generates interrupts at a constant frequency, known as the **system clock** (not to be confused with the CPU clock). The interrupts are called clock ticks, or simply **ticks**. The system clock is used to decide the current time, in order to bring the different processes in line chronologically e.g. disk handling, sound, delays etc.

The frequencies of the system clocks in different computers vary, but in general is between 10 and 100 ticks/s (Hz). In PC compatible computers running DOS (from the original PC to PS/2 and 486s), this clock frequency has been set to approximately 18.2 ticks/s or 55 ms/tick. The former value has been exactly defined in the floating point constant **MG.ticksPerSecond**.

The system clock can be used for measuring time intervals, but can also give absolute time, i.e. the time of day. When the computer is switched on, the system clock is usually initialized automatically so that the value 0 corresponds to midnight (provided that the computer has a battery operated real-time clock that functions properly).
Event Types

When the unit **MGClock** is imported into a program, a service routine is also installed for the clock interrupts. The event queue in **MGEv** is automatically updated from then on, with a clock event for each tick i.e. approximately 18 times/s. The generating of events may be stopped using the procedure **StopClockEvents** and started again with **StartClockEvents**.

This repetitive event type is called **clockTick**. The type may be used for cyclic events, or simply to keep track of the time of day.

Moreover, **MGClock** also makes it possible to initialize and supervise two other event types, namely **oneTimeAlarm** and **repeatedAlarm**. The former event type generates an event (an alarm) when the system clock has reached a specific time of day. The other type generates repeated events (alarms) using a desired interval. In both cases, up to 8 different alarms may be controlled (8 one-time alarms and 8 repeating alarms).

The one-time alarms can be used for supervising events, such as the "time-out" type etc. Repeating alarms can be used for periodical or intermittent events, i.e. those that do not need to be controlled at each tick of the system clock (updating of the hands of a clock, the collection of data from sensors, the movements in animated graphics, etc.).

Reading Clock Events

This is what a section of a program that reads clock events may look like (in this case, events not older than 1 second):

```pascal
VAR
  clockEv: MGClock.ClockEvent;
  evAge   : Integer32;

CASE MGEv.NextEventGen OF
  MGEv.clockGen:
    BEGIN
      MGEv.RdNextEvent(clockEv,evAge);
      IF evAge <= 2*MGClock.ticksPerSecond THEN BEGIN
```

Ref 28-2

**MGClock - Timer**

**MultiGraphics & Borland Pascal/Turbo Pascal**

**DATABITEN**
Using *set* expressions:

```pascal
IF MGClock.oneTimeAlarm IN clockEv.events THEN BEGIN
  IF 0 IN clockEv.oneTimeAlarms THEN ..
  IF 1 IN clockEv.oneTimeAlarms THEN ..
  ..
  IF 7 IN clockEv.oneTimeAlarms THEN ..
END;
```

Using *logical arithmetic* expressions:

```pascal
IF MGClock.logrepeatedAlarm AND clockEv.logEvents <> 0 THEN BEGIN
  IF 1 AND clockEv.logRepeatedAlarms <> 0 THEN ..
  IF 2 AND clockEv.logRepeatedAlarms <> 0 THEN ..
  ..
  IF 128 AND clockEv.logRepeatedAlarms <> 0 THEN ..
END;
END;
```

**Time Conversions**

*MGClock* also contains routines for reading the system clock directly (*ClockNow*) and converting from ticks to the time in hours, minutes and seconds, measured from midnight (*ClockToTime* and *ClockAtTime*). There’s no need for the event handling to be active when these routines are called (it is switched off by using *StopClockEvents*).

Here is how ticks of the system clock are read and converted to hours, minutes and seconds:

```pascal
VAR
  h,m,s: Word;
  ..
MGClock.ClockToTime(MGClock.ClockNow,h,m,s);
```

**Screen Clocks**

It is welcome to be able to continuously show the current time on the screen. With *MultiGraphics* and the unit *MGClock* it is a simple matter to integrate digital clocks as well as analog clocks in the same program. *MGClock* provides you with great freedom of choice when it comes to selecting the clock's appearance, size and method of keeping the clock updated.
Clocks are linked to windows and are installed just like new *capabilities* by using `MG.EnhanceWin`. It is therefore possible to show several clocks simultaneously on the screen (or on different screens for that matter), each in its own window.

### Digital Clocks

Digital clocks are linked to windows by using the capability `digiCap`. The linking is carried out using the call

```pascal
MG.EnhanceWin(win, MGClock.digiCap);
```

When the clock capability is initialized, the font of the digits is assigned to the current font, their color to white and the background to black. The printout is automatically centered in the clock's window and the time is presented as follows:

![Figure 1: Digital clock](image)

The background color, the digits' font and color can be chosen more freely by calling `DrawDigiClock` using the clock attribute data in a parameter of the type `DigiClockLook`. The digits can also be shaded. The clock is updated by calling `DrawDigits`.

Here is an outline of a program that shows a digital clock and updates the time using event handling.
VAR
digiWin : MG.Window;
digiLook: MGClock.DigiClockLook;
digitFnt: MG.Font;
clockEv : MGClock.ClockEvent;
evAge   : Integer32;
h,m,s   : Word;
.
MG.LoadFont(digitFnt,'sans.cbs');
MG.SetFont(digitFnt);
width:= ;
height:= ;
MG.CreateWin(digiWin,
0,0,
2*MG.StrWidth('00:00:00'),
2*MG.CurCharHeight);
MG.EnhanceWin(digiWin,MGClock.digiCap);
WITH digiLook DO BEGIN
  backCol  := MG.MatchingBackground(MG.cyan);
  digitFont:= digitFnt;
  digitCol := MG.MatchingForeground(MG.white);
  shadow   := TRUE;
END;
MGClock.DrawDigiClock(digiWin,digiLook);
REPEAT
  CASE MGEv.NextEventGen OF
    ..
    MGEv.clockEv: BEGIN
      MGEv.RdNextEvent(clockEv,evAge);
      IF (evAge < MGClock.ticksPerSecond) &
         (MGClock.clockTick IN clockEv.events) THEN BEGIN
        MGClock.ClockToTime(clockEv.ticks,h,m,s);
        MGClock.DrawDigits(digiWin,h,m,s);
        ..
      END; (*case*)
  UNTIL done;

Analog Clocks

Analog clocks are attached to windows using the capability anaCap. This is done by the call

**MG. EnhanceWin(win,MGClock.anaCap);**

Analog clocks are shown with a round clock-face whose radius is automatically adapted to the size of the window the clock has been attached to. The color and pattern of the clock-face, the color and pattern of the hands of the clock, as well as the color of the graduations on the clock-face, can be
chosen freely. Moreover, the hands of the clock and the graduations can be shaded to give added 3-D realism to the presentation. However, the second-hand is always drawn (due to speed considerations) using the drawing mode complement.

The appearance of the clock-face and the hands of the clock are defined using the data type **AnaClockLook**. The clock-face is drawn using **DrawAnaClock**. The hands of the clock (with any possible shading) are drawn using **DrawHands**. The second-hand need not be drawn.

Here is an outline of a program that shows the face of a clock and updates the time using event handling.

```pascal
VAR
  clockWin : MG.Window;
  clockLook: MGClock.AnaClockLook;
  clockEv  : MGClock.ClockEvent;
  evAge    : LongInt;
  h,m,s    : Word;

MGW.CreateWin(clockWin,0.3,0.3,0.4,0.4);
MG.EnhanceWin(clockWin,MGClock.anaCap);
WITH clockLook DO BEGIN
  shape        := MGClock.squareClock;
  backCol      := MG.MatchingForeground(MG.brown);
  backPat      := MGPat.bricks8;
  hourHandCol  := MG.MatchingForeground(MG.yellow);
  hourHandPat  := MGPat.GenPattern(MGPat.halftone6);
  minuteHandCol:= MG.MatchingForeground(MG.yellow);
  minuteHandPat:= MGPat.GenPattern(MGPat.halftone8);
  ticksCol     := MG.MatchingForeground(MG.yellow);
  shadow       := TRUE;
END; (*with*)
MGClock.DrawAnaClock(clockWin,clockLook);
```
REPEAT
  CASE MGEv.NextEventGen OF
  ..
    MGEv.clockEv: BEGIN
      MGEv.RdNextEvent(clockEv,evAge);
      IF (evAge < MGClock.ticksPerSecond) &
      (MGClock.clockTick IN clockEv.events) THEN BEGIN
        MGClock.ClockToTime(clockEv.ticks,h,m,s);
        MGClock.DrawDigits(clockWin,h,m,s);
      ..
      END; (*case*)
  UNTIL done;

Program Examples

Examples of the handling of clock events are found in GAClock2, and GAMouse. Two digital clocks are demonstrated in GADigi, while two analog clocks are shown in GAAna. Analog clocks are also displayed in GAClock1 (polling) and GAClock2 (event controlled).

Constants

<table>
<thead>
<tr>
<th>unitName</th>
<th>'MGClock'</th>
</tr>
</thead>
<tbody>
<tr>
<td>copyright</td>
<td>'DATABITEN'</td>
</tr>
<tr>
<td>clockActive</td>
<td>1160</td>
</tr>
<tr>
<td>invalidAlarm</td>
<td>1161</td>
</tr>
<tr>
<td>invalidTicks</td>
<td>1162</td>
</tr>
<tr>
<td>invalidTime</td>
<td>1163</td>
</tr>
</tbody>
</table>

Error codes for specific errors in the unit MGClock.

ticksPerDay
ticksPerHour
ticksPerMinute
ticksPerSecond

Long integer constants, which specify data for the system clock. For the sake of portability, these constants (or MG.ticksPerSecond) should be used instead of absolute values.
Ref 28-8  MGClock - Timer

clockTick     = 0;
oneTimeAlarm  = 1;
repeatedAlarm = 2;

Event types in MGClock used in set expressions.

allClockEvents =
[clockTick, oneTimeAlarm, repeatedAlarm];

A set constant (of BitSet16 type) that comprises all event types in MGClock.

logClockTick      = 1;
logOneTimeAlarm   = 2;
logRepeatedAlarm  = 4;

The same event types but used in logical arithmetic expression ("AND", "OR", "NOT", etc.)

logAllClockEvents = 1 + 2 + 4;

Constant (of type Word16) including all event types in MGClock and used in logical arithmetic expressions.

maxAlarm 7

The maximum number of one-time alarms and repeating alarms respectively that can be active simultaneously.

Data Types

Alarm = 0..maxAlarm;

Data types for one-time alarms and repeating alarms. Each alarm has a number from 0 to 7 (there is a number 2 for single alarms and another number 2 for repeating alarms). The number is used when the alarm is installed (SetOneTimeAlarm and SetRepeatedAlarm) and uninstalled (ResetOneTimeAlarm and ResetRepeatedAlarm). The same number is also returned when alarm events are generated.
AlarmSet = SET OF Alarm;

Set type (occupies 1 byte) for alarms.

ClockEvent = RECORD
  ticks : LongInt;
  gen  : EventGen;
  CASE INTEGER OF
    0: (events : BitSet16;
         oneTimeAlarms : AlarmSet;
         repeatedAlarms : AlarmSet;
         unused : ARRAY[0..8] OF Byte);
    1: (logEvents : Cardinal16;
         logOneTimeAlarms : Byte;
         logRepeatedAlarms : Byte;
         logUnused : ARRAY[0..8] OF Byte);
  END;

Data type for clock events. The first three fields are the same as those defined in MGEv.Event. The field events is interpreted using the constants clockTick, oneTimeAlarm and repeatedAlarm above. The variant field logEvents is instead interpreted using the constants logClockTick, logOneTimeAlarm and logRepeatedAlarm.

The field oneTimeAlarms specifies the set of one-time alarms that has been activated by the event (if the set is not empty, oneTimeAlarm is also included in events). The variant field logOneTimeAlarm is used if you prefer logical arithmetic expressions. The field repeatedAlarms gives the set of repeating alarms that has been activated by the event (if the set is not empty, repeatedAlarm is also included in events). You will find the same information in the logical arithmetic variant logRepeatedAlarm.

Clock events are special in that the event type clockTick is common for all events. If oneTimeAlarm or repeatedAlarm is included in an event, you have to decide which alarm or alarms that have been activated by inspecting the fields oneTimeAlarms and repeatedAlarms.
DigiClockLook = RECORD  
  backCol  : MG.DeviceColor;  
  digitFont: MG.Font;  
  digitCol : MG.DeviceColor;  
  shadow   : BOOLEAN;  
END;

Data types for defining the appearance of digital clocks (see DrawDigitClock). backCol specifies the clock window's background color, digitFont specifies the digits' font and digitCol the digits' color, shadow specifies if the digits are to be shaded (TRUE) or not (FALSE).

AnaClockShape = (squareClock, roundClock);  
AnaClockLook = RECORD  
  shape        : AnaClockShape;  
  backCol      : MG.DeviceColor;  
  backPat      : MG.Pattern;  
  hourHandCol  : MG.DeviceColor;  
  hourHandPat  : MG.Pattern;  
  minuteHandCol: MG.DeviceColor;  
  minuteHandPat: MG.Pattern;  
  ticksCol     : MG.DeviceColor;  
  shadow       : BOOLEAN;  
END;

Data types for defining the appearance of analog clocks (see DrawAnaClock). shape specifies whether the rectangular clock window will be filled (squareClock) or not (roundClock), backCol specifies the foreground color of the Clock-face and backPat its pattern, hourHandCol gives the foreground color of the hour hand and hourHandPat the pattern for the hour hand, minuteHandCol specifies the color of the minute hand and minuteHandPat its pattern, ticksCol gives the foreground color of the graduations and, finally, shadow specifies if the hands of the clock and the graduations are to be shaded (TRUE) or not (FALSE).
Variables

EP: SV.ErrProc;
Error procedure variable in the unit MGClock.

digiCap: MG.Capability;
Window capability for digital clocks. It is installed by calling MG.EnhanceWin.

anaCap: MG.Capability;
Window capability for analog clocks. It is installed by calling MG.EnhanceWin.

Procedures & Functions

Starting Clock Events

StartClockEvents

P StartClockEvents;
This procedure links an interrupt routine to the clock interrupts (interrupt vector 1CH on a PC) and installs an event service function for the clock in MGEv. From now on, each tick will generate a clock event, which is then placed in the event queue. All alarms installed are switched off.

The procedure is called automatically by the initializing section in MGClock, which means that StartClockEvents only needs to be called if the clock has previously been switched off using StopClockEvents.

SV.Result
clockActive the clock generator is already active
**StopClockEvents**

```
P StopClockEvents;
```

This procedure stops the clock generator and resets the interrupt routine (the interrupt vector 1CH on a PC), which was installed before `MGClock` was initialized.

`StopClockEvents` is called automatically by the termination routine in `MGClock`, which means the procedure need not be called manually when the program is terminated.

The procedure does nothing if a clock is not installed.

The result is always assigned to `SV.ok`.

**Installing Alarms**

**SetOneTimeAlarm**

```
P SetOneTimeAlarm(al : Alarm; 
atTicks: Integer32);
```

This procedure installs a *one-time alarm* with the number in `al` which generates an alarm event at the time `atTicks` (absolute time). Any previous alarm using the number in `al` is overwritten.

- **SV.Result invalidAlarm** the alarm number must be
  - `<= maxAlarm` (7)
- **invalidTime** the alarm time must be within 24 hours

```(* Set the alarm to 0 at midnight *)
MGClock.SetOneTimeAlarm(0,0);
(* Set alarm 1 at 8:15:30*)
MGClock.SetOneTimeAlarm(1,MGClock.ClockAtTime(8,15,30));
(*Set alarm 2, 2 minutes later*)
MGClock.SetOneTimeAlarm(2,(MGClock.ClockNow + 
  2*MGClock.ticksPerMinute) 
  MOD MGClock.ticksPerDay);```
ResetOneTimeAlarm

P ResetOneTimeAlarm(al: Alarm);

This procedure switches off any installed one-time alarm with the number in al.

SV.Result
invalidAlarm the alarm number must be <= maxAlarm (7)

SetRepeatedAlarm

P SetRepeatedAlarm(al          : Alarm;
                   tickInterval: Integer32);

This procedure installs a repeating alarm, having the number in al, which generates alarm events periodically using tickInterval as the tick range, starting from when the call is made. Any previous alarm using the number in al is overwritten.

SV.Result
invalidAlarm the alarm number must be <= maxAlarm (7)
invalidTicks the range must be > 0 and at most 24 hours later.

(* Set alarm 0 to approx. once/s *)
MGClock.SetRepeatedAlarm(0,MGClock.ticksPerSecond);
(* Set alarm 1 to approx. once/min *)
MGClock.SetRepeatedAlarm(0,MGClock.ticksPerMinute);

ResetRepeatedAlarm

P ResetRepeatedAlarm(al: Alarm);

This procedure switches off any possible repeating alarm installed having the number in al.
SV.Result
invalidAlarm the alarm number must be
<= maxAlarm (7)

Converting Time

**ClockToTime**

```pascal
PROCEDURE ClockToTime(    ticks  : Integer32;
                         VAR hours  : Word;    (*O*)
                         VAR minutes: Word;    (*O*)
                         VAR seconds: Word     (*O*));
```

This procedure converts the time \textit{ticks} to a clock time in hours, minutes and seconds. The procedure assumes that the system clock is correctly set (in PC's set to 0 at midnight).

\textbf{ClockToTime} does not use event handling in \textit{MGEv} which means that this procedure can be called even if the generating of clock events is switched off (\textit{StopClockEvents}).

The procedure does not set (due to speed considerations) error codes in \textit{SV}; the result of the call cannot be checked by using \textit{SV.Result}.

```pascal
VAR
  clockEv: MGClock.ClockEvent;
  evAge  : LongInt;
  h,m,s  : Word;
  ...
MGEv.RdNextEvent(clockEv,evAge);
IF evAge < MGEv.ticksPerSecond THEN
  MGClock.ClockToTime(clockEv.ticks,h,m,s);
  MGIO.WrWord(h,2);MGIO.WrChar(':');
  MGIO.WrWord(m,2);MGIO.WrChar(':');
  MGIO.WrWord(s,2);
END;
```
**ClockAtTime**

```pascal
F ClockAtTime(hours, minutes, seconds: Word): Integer32;
```

This function converts and returns a time in hours, minutes and seconds as ticks of the clock. The result assumes that the system clock is correctly set (0 at midnight for PC’s).

*ClockAtTime* does not use event handling in *MGEv* which means that this function can be called even if the generating of clock events is switched off (*StopClockEvents*).

The function does not set (due to speed considerations) error codes in *SV*; the result of the call cannot therefore be checked by using *SV.Result*.

```pascal
MGClock.SetOneTimeAlarm(3, MGClock.ClockAtTime(20, 30, 00));
```

**Current Time**

```pascal
F ClockNow: Integer32;
```

This function reads the system clock and returns the current time as clock ticks. The time is correct if the system clock has been properly initialized (set to 0 at midnight for PC’s).

*ClockNow* does not use event handling in *MGEv* which means that this procedure can be called even if the generating of clock events is switched off (*StopClockEvents*).

The function does not set (due to speed considerations) error codes in *SV*; the result of the call cannot therefore be checked by using *SV.Result*.

```pascal
MGClock.SetOneTimeAlarm(7, MGClock.ClockNow + 100);
```
Digital Clocks

**DrawDigiClock**

```pascal
P DrawDigiClock(win : Window;
                look: DigiClockLook);
```

This procedure initializes and draws the Clock-face of a digital clock in the window `win` with the appearance in accordance with `look`. The parameter `look`, of the type `DigiClockLook`, gives a specification of the clock-face's color, font as well as the color of the digits. It also specifies if the digits are to be shaded or not.

The window `win` must have an enhanced digital clocks capability. The capability is installed with:

```pascal
MG.EnhanceWin(win,MGClock.digiCap);
```

`DrawDigiClock` does not use event handling in `MGEv` which means that this procedure can be called even if the generating of clock events is switched off (`StopClockEvents`).

**SV.Result**

- `processWithoutWin` the calling process has no window
- `capDontExist` the window has no digital clocks capability
- `invalidWin` invalid window
- `invalidDeviceColor` invalid device color in `look`

as well as other errors that are reported from the unit `MG`.
VAR
digiWin : MG.Window;
digiLook: MGClock.DigiClockLook;
digitFnt: MG.Font;

..  
MG.LoadFont(digitFnt,'sans.cbs');
MG.SetFont(digitFnt);
MG.CreateWin(digiWin,

   0,0,
   2*MG.StrWidth('00:00:00'),
   2*MG.CurCharHeight);
MG.EnhanceWin(digiWin,MGClock.digiCap);
WITH digiLook DO BEGIN
   backCol := MG.MatchingBackground(MG.cyan);
   digitFont:= digitFnt;
   digitCol := MG.MatchingForeground(MG.white);
   shadow  := TRUE;
END;
MGClock.DrawDigiClock(digiWin,digiLook);

**DrawDigits**

P DrawDigits(win : Window;
   h,m,s: Word);

This procedure displays the current time in the window win using those attributes that have been installed with DrawDigiClock. The time is given in h hours (0..23), m minutes (0..59) and s seconds (0..59). To be more effective, only those digits that have been changed since the previous call are drawn.

The window win must have an enhanced digital clocks capability (digiCap).

DrawDigits does not rely on event handling in MGEv which means that this procedure can be called even if the generating of clock events is switched off (StopClockEvents).

SV.Result

processWithoutWin the calling process has no window
capDontExist the window has no digital clocks capability
invalidWin invalid window
invalidTime invalid time given
Ref 28-18    MGClock - Timer

MGClock.ClockToTime(MGClock.ClockNow,h,m,s);
MGClock.DrawDigits(digiWin,h,m,s);

Analog Clocks

**DrawAnaClock**

```
P DrawAnaClock (win : Window;
look: AnaClockLook);
```

This procedure draws the Clock-face of an analog clock in the window `win` with the appearance in accordance with `look`. The size of circular Clock-face is automatically adapted to suit the window's size. The parameter `look`, of the type `AnaClockLook`, gives a specification of the clock-face's color and pattern, the hour and second-hands color and pattern, the graduations color and any possible shading of these. The second-hand is always drawn using the drawing mode `complement`.

The window `win` must have an enhanced analog clocks capability. The capability is installed with:

```
MG.EnhanceWin (win, MGClock.anaCap);
```

`DrawAnaClock` does not use event handling in `MGEv` which means that this procedure can be called even if the generating of clock events is switched off (`StopClockEvents`).

**SV.Result**

- `processWithoutWin` the calling process has no window
- `capDontExist` the window has no analog clocks capability
- `invalidWin` invalid window
- `invalidDeviceColor` invalid device color in `look`

as well as other errors that are reported from the unit `MG`.
VAR
clockWin : MG.Window;
clockLook: MGClock.AnaClockLook;
..
MGW.CreateWin(clockWin,0.3,0.3,0.4,0.4);
MG.EnhanceWin(clockWin,MGClock.anaCap);
WITH clockLook DO BEGIN
  backCol      := MG.MatchingForeground(MG.brown);
  backPat      := MGPat.bricks8;
  hourHandCol  := MG.MatchingForeground(MG.yellow);
  hourHandPat  := MGPat.GenPattern(MGPat.halftone6);
  minuteHandCol:= MG.MatchingForeground(MG.yellow);
  minuteHandPat:= MGPat.GenPattern(MGPat.halftone8);
  ticksCol     := MG.MatchingForeground(MG.yellow);
  shadow       := TRUE;
END; (*with*)
MGClock.DrawAnaClock(clockWin,clockLook);

**DrawHands**

P DrawHands (win : Window;
              h,m,s: Word);

This procedure draws the hour, minute and second-hands, *if* the hands of the
clock have moved when compared to the previous call. The time is given in
h hours (0..23), m minutes (0..59) and s seconds (0..59).

The hands of the clock are drawn in the window using those attributes that
have been given by DrawAnaClock.

If you do not want to show the second-hand, it can be switched off by set-
ting s to a value outside the range of 0..59, e.g., 65535.

The window win must have an enhanced analog clocks capability
(anaCap).

**DrawHands** does not rely on event handling in MGEv which means that
this procedure can be called even if the generating of clock events is
switched off (StopClockEvents).
SV.Result
processWithoutWin the calling process has no window
capDontExist the window has no analog clocks capability
invalidWin invalid window
invalidTime invalid time given

VAR
  h,m,s: Word;
.. REPEAT
  MGClock.ClockToTime(MGClock.ClockNow,h,m,s);
  MGClock.DrawHands(clockWin,h,m,s);
UNTIL MG.KeyPressed;
The MGKbd unit allows the keyboard to use the event handler in MGEv. When MGKbd is activated, each key pressed (see below) will generate a time-stamped keyboard event that is placed in the shared event queue. Each keyboard event contains information about the point in time for the key pressing, the complete keyboard code for the key pressed (using the DOS character code + the scan code), as well as the status of the shift keys (Shift, Alt, Ctrl etc.) and the keys with toggle functions (CapsLock, NumLock and ScrollLock).

Keyboard events can be read, just like any other events, by calling MGEv.RdNextEvent or MGEv.RdMatchingEvent. They can also be read using RdEventKey and checked for by usingKeyPressedEvent. When MGKbd is activated, the normal keyboard routines in MG, MGIO and even the standard Read and ReadLn are also redirected to read keys pressed from the event queue.

Some Details

We have previously stated that keys being pressed generate keyboard events. To be more precise however, we should add that not every key pressed generates an event.

MGKbd uses BIOS to receive information about which keys have been pressed. First, BIOS converts the key codes in accordance with the national keyboard and BIOS does not allow the shift keys and toggle keys to generate events by themselves, this only
occurs when they are pressed in conjunction with "ordinary" keys. The key codes that are read by MGKbd have therefore already been modified.

Secondly, BIOS itself places all keys being pressed in a queue before they are reported. The keyboard buffer in BIOS contains at most the codes for the latest 15 keys that have been pressed and if more keys are pressed in quick succession, and the event handler does not have time to deal with these events sufficiently quickly, characters are then lost (BIOS beeps when the keyboard buffer is full).

Program Outline

No special measures need to be taken to activate event handling of the keyboard. All that is needed is to import the unit MGKbd. Keys being pressed can be read using MGEv.RdNextEvent or by standard reading procedures in MG, MGIO or the standard Read/ReadLn.

USES ..,MGKbd;
.. 
VAR  
    key  : MG.KbdCode; 
    kbdEv: MGKbd.KbdEvent; 
    evAge: LongInt; 
.. 
MGIO.RdKbd(key);
.. 
CASE MGEv.NextEventGen OF 
..  
    MGEv.kbdGen: BEGIN 
        MGEv.RdNextEvent(kbdEv,evAge);
        IF kbdEv.key.wrd = MGIO.keys.escl.wrd THEN ..
.. 
END; (*case*)

Program Examples

GAKbd gives some examples of event controlled keyboard handling. The program reads the keys pressed and prints the character code as well as the status of the shift keys and toggle keys. Keyboard emulation is also demonstrated.

See also GAEvnts, GAMouse and GAClock2 which give general examples of event handling.
Constants

unitName 'MGKbd'
copyright 'DATABITEN'

kbdActive 1170
kbdNotActive 1171

Error codes specific for the unit MGKbd.

kbdKey 0

This is the only event type that is generated by the keyboard. It is used in set expressions.

allKbdEvents [kbdKey]

This is a set constant with the set of possible keyboard events (that solely contain kbdKey).

logKbdKey 1

Analogous event type to be used in logical arithmetic expressions ("AND", "OR", "NOT", etc).

logAllKbdEvents logKbdKey

Logical arithmetic constant with the set of possible keyboard events (that solely contain logKbdKey).

logRightShift 1
logLeftShift 2
logCtrl 4
logAlt 8
logLeftCtrl $10
logLeftAlt $20
logSysReq $40

Constants representing shift keys to be used in logical arithmetic expressions (see logShifts field in KbdEvent) as an alternative to set expressions (see ShiftKey type). Example:
IF MGKbd.logAlt AND kbdEv.logShifts <> 0 THEN ..

logScrollLock 1
logNumLock 2
logCapsLock 4

Constants representing toggle keys to be used in logical arithmetic expressions (see logToggles field in KbdEvent) as an alternative to set expressions (see ToggleKey type). Example:

IF MGKbd.logCapsLock AND kbdEv.logToggles <> 0 THEN ..

Data Types

ShiftKey = (rightShift, leftShift, ctrl, alt, leftCtrl, leftAlt, sysReq);

Enumerated data types for the keyboard's *shift keys*. Each keyboard event contains information about which shift keys are held down.

ShiftKeySet = SET OF ShiftKey;

The set type for shift keys. Example:

IF MGKbd.alt IN kbdEv.shifts THEN ..

ToggleKey = (scrollLock, numLock, capsLock);

Enumerated type for *toggle keys*. Keyboard events contain information about which toggle keys that are activated.

ToggleKeySet = SET OF ToggleKey;

The set type for activated toggle keys. For example:

IF MGKbd.capsLock IN kbdEv.toggles THEN ..
KbdEvent = RECORD
  ticks          : Integer32;
  gen            : EventGen;
  CASE INTEGER OF
    0: (events    : BitSet16;
         shifts    : ShiftKeySet;
         toggles   : ToggleKeySet;
         key       : MG.KbdCode;
         unused    : ARRAY[0..6] OF Byte);
    1: (logEvents : Word16;
         logShifts : Byte;
         logToggles: Byte;
         logKey    : Word16;
         logUnused : ARRAY[0..6] OF Byte);
  END;
END;

Data types for keyboard events. The first three fields are the same as those defined in MGEv.Event. The field for events is interpreted using the constant kbdKey above. The variant field logEvents is instead interpreted using the constant logKbdKey.

The field shifts/logShifts specifies the set of shift keys pressed when the event takes place. The field toggles/logToggles gives the set of activated toggle keys. The field key/logKey contains the keyboard codes for the particular keys pressed (or combination of keys) that generated the event.

Note that the only possible value for the field events is [kbdKey]. The only possible value for the logEvents field is logKbdKey.

Variables

EP: SV.ErrProc;

Error procedure variable in MGKbd.
Procedures & Functions

Starting Keyboard Events

**StartKbdEvents**

```pascal
P StartKbdEvents;
```

This procedure installs an event service routine for keyboard events and generates thereafter a keyboard event for each key pressed (that gives a keyboard code from BIOS).

The procedure redirects `MG.RdKey` and `MG.KeyPressed` to read the keyboard via the event queue instead of reading the keyboard directly. This means that both the error handling in `SV` and the text routines in `MGIO`, plus the standard `Read` and `ReadLn`, etc. from now on read the keyboard via the event handler in `MGEv`.

`StartKbdEvents` is called automatically when the unit `MGKbd` is initialized, which means that this procedure only needs to be called if the event handling has previously been switched off using `StopKbdEvents`.

```
SV.Result
kbdActive        event generating already activated
```

**StopKbdEvents**

```pascal
P StopKbdEvents;
```

This procedure stops the generating of keyboard events and resets the previous routines (BIOS). `MG.RdKey` and `MG.KeyPressed` revert to the previous keyboard routines (BIOS). When the call is made, the error handling in `SV` and the keyboard handling in `MGIO`, plus the standard `Read` and `ReadLn`, etc. are also reset.

The procedure is called automatically by the termination routines in `MGKbd` which means that `StopKbdEvents` does not need to be called manually when terminating a program.

The result is always `SV.ok`.
Reading Keyboard Events

**RdEventKey**

```pascal
P RdEventKey(VAR key: MG.KbdCode (*O*));
```

This procedure waits for and reads the next keyboard event (keys being pressed) from the event queue. The key code is returned in the variable parameter `key`.

*N.B. that events from other event generators that precede the key event are ignored and erased from the queue.*

Because `MG.RdKey`, `MGIO.RdKbd`, etc., are automatically redirected to read from the event queue when `MGKbd` has been activated (at initialization or by using `StartKbdEvents`), `RdEventKey` is seldom called directly.

**SV.Result**

`kbdNotActive` — event generating from the keyboard not activated

**KeyPressedEvent**

```pascal
F KeyPressedEvent: BOOLEAN;
```

This function checks to see if there is a keyboard event in the event queue or not.

*N.B. that events from other event generators are ignored and removed from the queue.*

After the call is made, there are therefore only two alternatives; either the queue is empty or the next event is a keyboard event.

Because `MG.KeyPressed` is automatically redirected to read from the event queue when `MGKbd` has been activated (at initialization or by using `StartKbdEvents`), `KeyPressedEvent` is seldom called directly.
Emulating the Keyboard

```
PG EmulateKbdEvent(key : MG.KbdCode;
                shifts : ShiftKeySet;
                toggles: ToggleKeySet);
```

This procedure emulates a keyboard event by sending the key code in `key`, the status for the shift keys in `shifts`, and the status for the toggle keys in `toggles`, to the event queue. The event can be read in the same way as a genuine keyboard event.

```
SV.Result kbdNotActive the keyboard event generator is not active

MG.EmulateKbdEvent(MGIO.keys.escl,
                    [MGKbd.alt,MGKbd.ctrl],
                    [MGKbd.capsLock]);
```
The mouse is a *pointing* input device with the addition of one or more mouse buttons for making *choices*. With a mouse, it is easier to draw and handle graphic objects (mark, choose, move, scale, etc.), as well as make menu choices, etc. By combining mouse support with the keyboard, graphics programs are provided with a modern and user friendly interface.

**MGMouse** contains several useful procedures. Here is a sample of the things it can do:

- report the mouse's position,
- report the status of the buttons, i.e. which mouse buttons are being held down,
- report the keyboard shift keys status (*Shift*, *Alt*, *Ctrl*) at the time of mouse events,
- generate mouse events for the event queue,
- emulate mouse events via programs and
- draw a mouse pointer on the screen that mirrors the mouse's movements (*mouse tracking*).

**MGMouse** sees to it that the event handler in **MGEv** registers all mouse events (mouse movements, shift keys status, button pressing as well as its own event types). When the mouse is moved, events are generated at a frequency of between 10 and 100 times per second. These mouse events can be read by calling **MGEv.RdNextEvent**. The mouse's position and the status of the buttons can also be read directly without going via the event queue.
MGMouse also takes care of all **tracing** of the mouse pointer. This prevents clashes between the mouse pointer and other graphics in **MultiGraphics**, which would otherwise be the case if **MultiGraphics** allowed the mouse drivers (MOUSE.COM, MOUSE.SYS) to draw the pointer. The in-built pointer handling in **MGMouse** gives even greater freedom of choice of the type and appearance of the pointer. If the computer has two screens connected to it, we can choose for ourselves which screen the mouse will run on. The pointer handling also makes it possible to use the mouse on graphics devices that are unsupported by the mouse driver.

**MGMouse** can use symbols (see "MG - Symbols") with or without masks and even multi-colored images (see "MGImg - Images") as mouse pointers. You can even use animated pointers!

The mouse's current mode, **the mouse mode**, (pointer type, pointer color, the mouse's sensitivity etc.) may be saved and then reset. This makes it simple to write general routines that modify the mouse mode and restore it to the original mode afterwards.

**MGMouse** can also **emulate** mouse events. One common use for this is to emulate the mouse using the keyboard. This increases the portability of graphics programs since there are still many PC's that have no mouse support. Moreover, the user can then choose for her/himself which method of input he or she prefers (there are both "mouse haters" as well as "mouse lovers" around).

### Mouse Requirements

Under **DOS**, **MGMouse** only works with a mouse that is Microsoft compatible (most mice for PC's are included among these nowadays). The accompanying mouse driver (normally MOUSE.SYS, MOUSE.COM or MOUSE.EXE) must always be loaded.

**Remember, a mouse driver must be loaded if you do not intend to emulate the mouse only!**

The mouse can have one button (unusual), two buttons (more common) or three buttons (do occur). Pure mouse emulation gives you a "mouse" with three buttons. We recommend moderation with regard to using mouse buttons. For most users, it is very confusing to keep in mind which mouse button is to be used in a particular situation. Experience shows that it is often sufficient to use the left mouse button together with single and double
clicking. One further argument in favor of moderation is the increased portability that this allows. Macintoshes, for example, have only one mouse button.

**MGMouse** uses the following mouse driver functions: 0, 4, 7, 8, 12, 15 and 19. Older mouse drivers also fully support these functions.

### Mouse Event Types

The event types that are generated are mouse movements, pressing the mouse buttons (the left, right or middle button), releasing the mouse buttons and double-clicking. These standard events are numbered from 0 to 9 (see "Constants" below). Event types 10..15 are not used and can be used to define your own event types (see "Mouse Event Procedures").

By *double-clicking* is meant pressing and releasing a specific mouse button twice in quick succession. The releasing must normally occur within 1/2 second of the pressing in order to be registered as a double-click (the time can be changed with **SetDoubleClickInterval**).

### Types of Pointers

Normally, the mouse's movements are shown on the screen with the help of a pointer symbol, the *mouse pointer*. This automatic utility is called *mouse tracking*. In the broader sense, a pointer can also be any symbol (icon) that is moved across the screen in conjunction with mouse movements.

**MGMouse** handles all pointer presentation by itself. The standard mouse pointer in the mouse driver is never used. There are several reasons for this, some of the most important are that the technique in **MultiGraphics**:

- gives greater freedom in choosing types, size and appearance of the pointer,

- makes it possible to track a mouse pointer even on devices that are not supported by the mouse drivers,

- makes it possible, when using double monitor screens, to choose which screen the pointer is to be shown on,
Ref 30-4  

MGMouse - Mouse Support

- avoids collisions between the mouse pointer and the program's own graphics when competing for the monitor screen.

MGMouse can handle three types of pointers, namely:

- symbol pointers,
- masked symbol pointers and
- image pointers.

The current pointer type depends on how the pointer is installed - by SetSymbolPointer (symbol type), SetMaskPointer (masked symbol type) or SetImagePointer (image type).

The Default Mouse Pointer

The default pointer is a masked symbol pointer showing an arrow pointing upwards to the left.

The foreground color is set to the maximum device color (white) and the background device color is 0 (black).

Symbol Pointers

Symbol pointers are installed by SetSymbolPointer and it is likely that you will use the standard symbol library of the current device. Symbols 64 - 79 are reserved for mouse pointers and honored with their own symbolic names by MGMouse, namely:

leftUpArrow0, upArrow0, downArrow0, leftArrow0, rightArrow0, rectCross0, diagCross0, vertBar0, checkMark0, hand0, hourGlass0, vertArrows0, horArrows0, diagDownArrows0, diagUpArrows0 and crossArrows0.

You may use these names directly when calling SetSymbolPointer. For more information see "Variables" below and "MG - Symbols".

Symbol pointers may also be obtained from other bit mapped symbol libraries of the type MG.Symbols (usually loaded from disk).

Since fonts are special types of symbols, even bitmapped characters can be used as mouse pointers.

Symbol pointers can be drawn using all the drawing modes in Multi-
Graphics. Normally, complement, replace or stamp (see below) are used. The device color for the pointer's foreground and background colors can be chosen freely. The pointer is drawn with the symbol's hot spot at the mouse's position.

Pointer Symbols with Masks

Symbol pointers with mask are installed by SetMaskPointer and you will most likely use masked symbols from the default symbol library of the current device. Symbols 64 - 79 in cooperation with 192 - 207 are reserved for mouse pointers and honored with their own symbolic names by MGMouse, namely:

leftUpArrow, upArrow, downArrow, leftArrow, rightArrow, rectCross, diagCross, vertBar, checkMark, hand, hourGlass, vertArrows, horArrows, diagDownArrows, diagUpArrows and crossArrows.

You may use these names directly when calling SetMaskPointer. For more information see "Variables" below and "MG - Symbols".

Masked symbol pointers may also be obtained from normal bit mapped symbol libraries of the type MG.Symbols (usually loaded from disk). Two symbols are used for each pointer - one for the pointer and another for the mask. A masked symbol is drawn in two steps:

1. The mask symbol is drawn using the drawing mode mask, which means that the background is protected where the mask is "set to 1", and is erased where the mask is "set to 0".

2. The pointer symbol is drawn using the drawing mode mix.

Normally, the mask is chosen so that it protects the area around the pointer symbol, except for the contour lines which make the pointer visible no matter what the background.

Since fonts are special types of symbols, even bitmapped characters can be used as mouse pointers.

The default mouse pointer is a masked symbol pointer (arrow pointing upward to the left).

Masked pointers are slower to be drawn in comparison to simple pointer symbols, but they fill an important function especially with monochrome graphics devices. On color screens you can often use the drawing mode stamp and choose the pointer symbol's
color so that it always stands out.

Image Pointers

Unique for MultiGraphics is the ability to use images of the type MGImg.Image as mouse pointers. This opens up eye-catching possibilities, of which the following examples are but a few:

- multi-colored pointers,
- drawn, scanned or digitalized pointers,
- icons,
- pointer animation.

Since mouse pointers, in general, are rather small, images which contain a whole chart full of pointers are normally used. The desired pointer is specified using its position on the image as well as its size (of course, one image may contain a single pointer). It is quite easy to create a chart of pointers using an ordinary paint program (Paintbrush, etc.).

In contrast to pointer symbols, image pointers need not be monochrome. An image pointer is monochrome only if the image itself is monochrome. Monochrome mouse pointers are drawn using the pointer's current foreground and background colors.

By changing between a series of image pointers, you can produce animated mouse pointers. Since images in MultiGraphics are implemented with the help of virtual screens, there is nothing to prevent the image pointer from being modified using ordinary drawing commands during the execution of the program!

Drawing Modes for Mouse Pointers

Drawing modes for symbol and image pointers can be chosen freely. Normally, however, complement, replace or stamp is used.

The fastest of these is complement, which also has the advantage that the pointer is always visible no matter what the screen background. The disadvantage is, however, that the mouse's color changes with the background.

The stamp mode makes the screen background visible where the pointer is "set to 0" and always gives the pointer the same color no matter what the
background. If, however, the mouse pointer should find itself up against a background with the same color, it will not be seen. The problem is especially crucial when using monochrome screens. It is recommended here to use masked pointer symbols instead.

**Automatic Protection of the Mouse Pointer**

The tracking of the mouse pointer is done in parallel with the application drawing its own graphics. Of course, there is a potential risk of collisions here. At the initialization of the mouse therefore, routines are installed that protect the mouse pointer from being overwritten by other graphics operations in **MultiGraphics** and vice versa.

The protection works by having **MGMouse** automatically check if a graphic operation in **MultiGraphics** interferes with the mouse pointer. In that case the mouse pointer is switched off during the graphic operation and thereafter switched on again.

> It is imperative that the mouse handler is activated before any new windows are created in the mouse screen. Otherwise the mouse pointer protection will not work with these windows!

We can also switch off the protection completely and take over the responsibility for protection of the mouse pointer. The program has to switch off the mouse pointer when there is a risk of collision by using **HidePointer** and then switch it on again using **ShowPointer**.

**The Mouse Pointer's Visibility**

The pointer's *visibility* in **MGMouse** may be set to *fully visible* or to various "degrees" of invisibility. Each call to **HidePointer** decreases the visibility by one step, while calling **ShowPointer** increases the visibility by one step until the pointer is fully visible. **HidePointer** always makes the pointer invisible while a call to **ShowPointer** does not ensure that the pointer becomes visible on the screen, it merely becomes "less " invisible. Only when the pointer reaches the stage of *fully visible* will it be shown on the screen. This makes it possible to nest calls of **HidePointer** and **ShowPointer** without the latter procedure necessarily making the pointer visible.
Saving the Mouse Mode

The current *mouse mode* can be read and saved in a variable of the type `MouseMode`. This is done using the procedure `GetMouseMode`. The mouse mode may be reset later with `SetMouseMode`.

Mouse Event Procedures

This section is only of interest for advanced users.

A *mouse event procedure* is a procedure of the type `MouseEventProc` which when installed (with `SetMouseEventProc`) is automatically called on every mouse event.

```pascal
MouseEventProc =
  PROCEDURE(VAR ev : MouseEvent; (*IO*)
            pointerBusy: BOOLEAN);
```

The parameter `ev` contains the mouse event that initialized the call. The data in `ev` can be read and, if need be, modified by the mouse event procedure. Of special interest here, is that the event procedure can add user defined event types (10..15) to the field `ev.events`. The field `ev.user` is also free to be used to store user data about the event. The modified data in the `ev` variable is placed in the event queue and may later be read using `MGEv.RdNextEvent`, etc.

Mouse event procedures may be used to change the mouse pointer's appearance and color as a function of the current position of the mouse. They can also be used to create user mouse event types within the range of 10..15 and provide mouse events with your own specific data. One common usage is to generate events when the mouse crosses window boundaries on the screen and use the `user` field to store information about which window...
the mouse is in at the moment. This information can be used to open menus, give information, and to ask questions, etc.

If a mouse event procedure performs operations that modify the mouse pointer, this must only be done when the parameter `pointerBusy` is `FALSE`.

A mouse event procedure should be very fast (mouse events can occur 10 - 100 times/s) and it has limited stack resources to its disposal.

The current window of a mouse event procedure is the same as that of the mouse (which is created by `StartMouseEvents`).

**Program Outline**

With the help of a program outline, we will show you the most important steps to follow when using the unit `MGMouse`:

```pascal
VAR
  mousePtr: MGMouse.SymbolPointer;
  pointers: MG.Symbols;
  mouseEv : MGMouse.MouseEvent;
  evAge : LongInt;
  buttons: MGMouse.ButtonSet;
  mouseX,
  mouseY : INTEGER;

.. (* Initialize and select the mouse screen before any new windows are created*)
MGMouse.StartMouseEvents(MG.CurScr);
(* Now you can create new windows *)
..
(* Show the default pointer *)
MGMouse.ShowPointer;
..
(* Choose another masked mouse pointer *)
MGMouse.SetMaskPointer(MGMouse.hourGlass);
..
(* Make a new mouse pointer *)
MG.LoadSymbols(pointers,'POINTERS.SBS',FALSE);
WITH mousePtr DO BEGIN
  symbs := pointers;
  symbol := 3;
  drwMode := MG.stamp;
  foreground:= MG.MatchingForeground(MG.lightRed);
  background:= MG.MatchingBackground(MG.black);
END;
```
MGMouse.SetSymbolPointer(mousePtr);

(* Read mouse events *)
MGEv.RdNextEvent(mouseEv,evAge);
IF evAge < maxDelay THEN ..

(* Read current data directly from the mouse driver *)
MGMouse.GetMouseWinInfo(buttons,mouseX,mouseY);

(* Terminate *)
MG.SetScrWin;
MGMouse.StopMouseEvents;
MG.CloseDevice(..);

Program Examples

**GAEvents** shows you how to use pointer tracking, pointer protection, supervising the mouse's position and the status of the buttons. **GAEvents** also gives an example of using the keyboard direction keys for mouse emulation.

**GADSyms** shows the default symbols of the current device used as mouse pointers.

**GAMouse** illustrates using the different types of pointer and drawing modes, user mouse event procedures to change the pointer's appearance, how the mouse mode can be saved and reset, drawing pointers, etc.

**GAAAniPtr** displays animated mouse pointers.

**GAHit** shows automatic detection of windows on screen, automatic exchange of the mouse pointer and event generating when changing windows. In the example, a user mouse event procedure is used.

**GAClock1** illustrates a polling technique for handling the mouse, keyboard and system clock. **GAClock2** shows event handling using a mouse.

**Creator** displays symbol pointers as well as image pointers.
Constants

unitName "MGMouse"
copyright "DATABITEN"

invalidMouse 1180
invalidPos 1181
invalidPointer 1182
invalidSens 1183

Error codes, specific for the unit MGMouse.

mouseMoved 0
leftButtonPressed 1
leftButtonReleased 2
rightButtonPressed 3
rightButtonReleased 4
middleButtonPressed 5
middleButtonReleased 6
leftButtonDoubleClick 7
rightButtonDoubleClick 8
middleButtonDoubleClick 9

Pre-declared event types for the mouse. These are included as elements in the field MouseEvent.events of the type BitSet. The numbers within the range 10..15 may be used for user defined event types that are generated by mouse event procedures.

allMouseEvents = [0..15];

A set constant of the type BitSet which contains all mouse event types (also user defined).
logMouseMoved = 1;
logLeftButtonPressed = 2;
logLeftButtonReleased = 4;
logRightButtonPressed = 8;
logRightButtonReleased = $10;
logMiddleButtonPressed = $20;
logMiddleButtonReleased = $40;
logLeftButtonDoubleClick = $80;
logRightButtonDoubleClick = $100;
logMiddleButtonDoubleClick = $200;

Matching logical arithmetic event types for the mouse to be used with the Word field MouseEvent.logEvents. The values $400, $800, $1000, $2000, $4000 and $8000 may be freely used for user defined event types that are generated by mouse event procedures.

logAllMouseEvents = $FFFF;

Logical arithmetic constant which contains all mouse event types (also user defined).

logLeftButton = 1;
logRightButton = 2;
logMiddleButton = 4;

Constants that are used in logical arithmetic expressions to specify which mouse buttons are being held down. A one button mouse uses only logLeftButton, a two buttons mouse uses logLeftButton and logRightButton, while a three buttons mouse uses logMiddleButton as well. Also see the enumeration type Button for a set oriented variant of this. Example:

IF MGMouse.logLeftButton AND mouseEv.logButtons <> 0 THEN ..

logRightShift = 1
logLeftShift = 2
logCtrl = 4
logAlt = 8
logLeftCtrl = $10
logLeftAlt = $20
logSysReq = $40

Constants representing shift keys to be used in logical arithmetic expres-
sions (see logShifts field in MouseEvent) as an alternative to set expressions (see ShiftKey type). Example:

IF MGMouse.logAlt AND mouseEv.logShifts <> 0 THEN ..

NullMouseEvent;

Represents a non-existent mouse event procedure of the type MouseEventProc.

Data Types

Button =
  (leftButton,
   rightButton,
   middleButton);

Enumerated data types that are used in set expressions to specify which mouse buttons are being held down. A mouse with a single button uses only leftButton, a two buttons mouse uses leftButton and rightButton, while a three buttons mouse uses middleButton as well.

ButtonSet = SET OF Button;

A set type for mouse buttons pressed. For example:

IF MGMouse.leftButton IN mouseEv.buttons THEN ..

ShiftKey = (rightShift,leftShift,
  ctrl,alt,leftCtrl,leftAlt,
  sysReq);

Enumerated data types for the keyboard's shift keys. Each mouse event contains information about which shift keys are held down.

ShiftKeySet = SET OF ShiftKey;

The set type for shift keys. Example:

IF MGMouse.alt IN mouseEv.shifts THEN ..
MouseEvent = RECORD
  ticks : Integer32;
  gen  : EventGen;
END;

Data types for mouse events. The first three fields are identical in size and order to the fields in MGEv.Event. The field gen is assigned the value MGEv.mouseGen when mouse events occur. The field events is interpreted using the constants mouseMoved..middleButtonDoubleClick with the addition of any possible user defined event types. The corresponding logEvents variant field is interpreted using the logical arithmetic constants logMouseMoved..logMiddleButtonDoubleClick.

The field buttons/logButtons specifies the set of mouse buttons pressed at the time of the event. The shifts field specifies the set of keyboard shift keys (Shift, Alt and Ctrl) pressed. The fields scrX and scrY give the mouse's position in screen window coordinates at the event. Finally, user can be used by user mouse event procedures (type MouseEventProc) in order to report its own data in the mouse events (note that user comprises 5 bytes, i.e. it can contain a pointer (4 bytes) to a more comprehensive data structure).

MouseEventProc =
  PROCEDURE(VAR ev : MouseEvent; (*IO*)
             pointerBusy: BOOLEAN);
mouse event procedures can be used, for example, to change the appearance or color of the pointer when the mouse finds itself within a particular area of the screen, or to modify a mouse with user event types (10..15 in Event.events). In the latter case, the ev parameter may be modified in the fields events and user. Before the mouse event procedure performs operations that modify the mouse pointer, pointerBusy must be checked. This also applies to calls made to standard routines in MultiGraphics that use locking/unlocking procedures and in those cases where the mouse pointer is protected.

Mouse event procedures are installed using SetMouseEventProc.

MousePointerType = (symbolMousePointer,
maskMousePointer,
imageMousePointer); Enumerated types for those mouse pointer types that are accessible in MGMouse. symbolMousePointer is installed by SetSymbolPointer, maskMousePointer is installed by SetMaskPointer and finally, imagePointer, is installed by using SetImagePointer.

SymbolPointer = RECORD
  symbs     : MG.Symbols;
  symbol    : Word;
  drwMode   : MG.DrawMode;
  foreground: MG.DeviceColor;
  background: MG.DeviceColor;
END;

Data types for pointer symbols. The field symbs specifies the pointer's symbol libraries, symbol the desired symbol in that library, drwMode the pointer's drawing mode, foreground the pointer's foreground device color and background the background device color.

MaskPointer = RECORD
  symbs     : MG.Symbols;
  symbol    : Word;
  maskSymbol: Word;
  foreground: MG.DeviceColor;
  background: MG.DeviceColor;
END;

Data types for masked pointer symbols. The field symbs specifies the
pointer's symbol libraries, symbol the desired symbol in that library, maskSymbol the appropriate masking symbol, foreground the pointer's foreground device color and background the background device color.

ImagePointer = RECORD
  img : MGImg.Image;
  leftSkip, bottomSkip : INTEGER;
  width, height : INTEGER;
  hotX, hotY : INTEGER;
  drwMode : MG.DrawMode;
  foreground : MG.DeviceColor;
  background : MG.DeviceColor;
END;

Data types for image pointers. The field img specifies an image with one or more pointers, leftSkip and bottomSkip the pointer's lower, left corner in the image, width and height the pointer's width and height, hotX and hotY the position of the hot spot (drawing point) in the pointer, drwMode the drawing mode for the pointer, foreground and background the pointer's foreground and background device colors respectively.

MouseMode = RECORD
  mouseScr : MG.Screen;
  doubleClickInterval : Word;
  doubleSpeedThreshold : Word;
  horSens, vertSens : Word;
  pointerProtected : BOOLEAN;
  protectionWin : MG.Window;
  limitationWin : MG.Window;
  EventProc : MouseEventProc;
  CASE pointerType : MousePointerType OF
    symbolMousePointer : SymbolPointer;
    (symbolPtr : SymbolPointer);
    maskMousePointer : MaskPointer;
    (maskPtr : MaskPointer);
    imageMousePointer : ImagePointer;
    (imagePtr : ImagePointer);
END;

Data type that define the current mode of the mouse pointer. It is used as a parameter type in GetMouseMode and SetMouseMode.
**Variables**

**EP**: SV.ErrProc;

Error procedure variable for the unit MGMouse.

**VAR**

leftUpArrow0,  
upArrow0, downArrow0,  
leftArrow0, rightArrow0,  
rectCross0, diagCross0,  
vertBar0,  
checkMark0,  
hand0,  
hourGlass0,  
vertArrows0, horArrows0,  
diagDownArrows0, diagUpArrows0,  
crossArrows0 : SymbolPointer;

Initialized variables normally used without modification when installing mouse pointers of symbol type (*SetSymbolPointer*).

**VAR**

leftUpArrow,  (* The default mouse pointer *)  
upArrow, downArrow,  
leftArrow, rightArrow,  
rectCross, diagCross,  
vertBar,  
checkMark,  
hand,  
hourGlass,  
vertArrows, horArrows,  
diagDownArrows, diagUpArrows,  
crossArrows : MaskPointer;

Initialized variables normally used without modification when installing mouse pointers of masked symbol type (*SetMaskPointer*). leftUpArrow is the default symbol used as mouse pointer if not changed.
Mouse Data

**MouseExist**

F MouseExist: BOOLEAN;

This function returns TRUE if there is a mouse driver installed and this is responding to calls. The function reflects the situation at the initialization of the unit and the result is not changed during run-time, even if an existing mouse is removed or a new mouse installed.

**MouseExist** does not use error handling in the unit SV.

IF MGMouse.MouseExist THEN
   MGMouse.StartMouseEvents(MG.CurScr);
.

**MouseButtons**

F MouseButtons: Word;

This function returns the number of mouse buttons on an installed mouse.

**MouseButtons** does not use error handling in the unit SV.

Activating the Mouse

**StartMouseEvents**

P StartMouseEvents(mouseScr: Screen);

This procedure installs an event service routine for mouse events and attaches the mouse to the screen that is specified by mouseScr.

**StartMouseEvents** is not executed automatically when initializing the unit MGMouse, which means that this procedure must be called before other
mouse routines are used and mouse events are supervised. The procedure opens a window for the mouse in the screen that is given by `mouseScr`. The mouse window comprises the whole screen. `StartMouseEvents` does not change the current window however.

`StartMouseEvents` should be called as one of the first actions of a program and always before any new windows in the mouse screen are created. This is most important! Otherwise the automatic mouse pointer protection will not work. The procedure is only allowed to be called for one screen in a program. If, therefore, a program uses two different screen devices, only one of them is allowed to use the mouse pointer. This cannot be changed during execution!

By the call, the default mouse pointer (arrow pointing upwards to the left) is installed but not made visible (you have to call `ShowPointer` for this).

The mouse driver (normally MOUSE.SYS, MOUSE.COM or MOUSE.EXE) must be installed so that the mouse handling in MultiGraphics can function. The mouse emulation (see `EmulateMouseEvent`), however, will work even without mouse and mouse driver.

SV.Result invalidMouse mouse driver or mouse is missing as well as other errors in MGErr.

```pascal
MG.SetDevice(driver);
MG.SetVisiScr;
MGMouse.StartMouseEvents(MG.CurScr);
```

**StopMouseEvents**

```pascal
P StopMouseEvents;
```

This procedure stops the generating of mouse events and the mouse tracking. It closes the mouse window and resets the external mouse driver (if present) to its standard setting. If the mouse has not been activated with `StartMouseEvents`, then nothing is done.

`StopMouseEvents` must always be called before the program is terminated, whether this is done normally (CloseDevice) or because of run time errors. In the latter case, it is advisable to place `StopMouseEvents` in the program's termination routine.
Since `StopMouseEvents` closes the mouse window, this must not be the current window when the call is made to the procedure. To avoid this just call `MG.SetScrWin` first.

`SV.Result` always returns `SV.ok`.

```pascal
MGMouse.StartMouseEvents(MG.CurScr);
...
MG.SetScrWin;
MGMouse.StopMouseEvents;
MG.CloseDevice(driver);
```

### Mouse Modes

#### GetMouseMode

```pascal
P GetMouseMode(VAR mode: MouseMode (*O*));
```

This procedure reads the current `mouse mode` and returns this in the variable `mode`. The mouse mode contains information about the mouse pointer's type, drawing mode, color, if the mouse pointer is protected or not, which part of the screen is protected, the boundaries for the mouse's freedom of movement, etc. (see the data type `MouseMode`).

By calling `GetMouseMode` before a section of the program that modifies the mouse mode and then directly after this call `SetMouseMode`, the previous mouse mode can be restored. This technique is very useful in writing general mouse based routines. N.B. that the position of the mouse position and the status of the buttons are not considered a part of the mouse mode.

`SV.Result`  
`invalidMouse`  
mouse is missing or invalid driver
VAR
    oldMouseMode: MGMouse.MouseMode;
    mousePtr    : MGMouse.SymbolPointer;

    (* Save the mouse mode *)
    MGMouse.GetMouseMode(oldMouseMode);
    ..
    MGMouse.SetSymbolPointer(mousePtr);
    ..
    (* Restore the old mouse mode *)
    MGMouse.SetMouseMode(oldMouseMode);

SetMouseMode

P SetMouseMode(VAR mode: MouseMode (*I*)));

This procedure sets the mouse mode that is specified by the variable parameter mode. Normally, mode has previously been filled with data by a call to GetMouseMode.

Due to the considerations of speed, mode is a variable parameter but is not modified when the call is made. It is however appropriate here to give a special word of warning about the EventProc field. If this field has a value <> NullMouseEvent, MGMouse assumes that the value represents a valid mouse event procedure. If this is not so, then the program will almost certainly crash!

SV.Result
invalidMouse           mouse driver or mouse is missing
MGErr.invalidDrawMode  invalid drawing mode
MGErr.outOfMem         not enough memory to protect
                       the screen background
invalidPointer         invalid symbol
MGImg.invalidImage     invalid image variable
MGErr.invalidWin       invalid protection window/limiting
                       window
invalidSens            invalid sensitivity value
CurMouseDev

F CurMouseDev: Device;

This function returns the graphics device that the mouse is attached to, i.e. the device that contains the mouse screen installed using StartMouseEvents.

It returns MG.nullDevice if StartMouseEvents has not been called.

SV.Result is always assigned to SV.ok

CurMouseScr

F CurMouseScr: Screen;

This function returns the screen that the mouse is attached to, i.e. the same screen that has been installed with StartMouseEvents.

Returns MG.nullScreen if StartMouseEvents has not been called.

SV.Result is always assigned to SV.ok

CurMouseWin

F CurMouseWin: Window;

This function returns the window that the mouse is attached to, i.e. the full screen sized window that was created in the mouse screen when the call was made to StartMouseEvents.

Returns MG.nullWindow if StartMouseEvents has not been called.

SV.Result is always assigned to SV.ok
Mouse Event Procedures

SetMouseEventProc

```
P SetMouseEventProc(EventProc: MouseEventProc);
```

This function installs a user mouse event procedure that is called at every mouse event. The procedure, which must be of the type `MouseEventProc` (see "Data Types" above), has as its input parameters the current mouse event of the type `MouseEvent` and a flag which specifies if the mouse pointer is in the process of being updated or not. The call to `EventProc` is made before the pointer is drawn when doing mouse tracking.

If the current parameter is `NullMouseEvent` then no mouse event procedure is installed. This is also the case when initializing the unit `MGMouse`. On the other hand, `StartMouseEvents` or `StopMouseEvents` do not change a previously installed mouse event procedure.

`SV.Result` is always assigned to `SV.ok`.

```
CONST
  winChanged = 10;
  maxWinIdx = 3;

VAR
  winHandle : ARRAY[1..maxWinIdx] OF MG.WindowHandle;
  mousePtr  : ARRAY[0..maxWinIdx] OF MGMouse.SymbolPointer;
  oldHandles: MG.WindowHandleSet;
```
(*$F*$)
PROCEDURE WinDetect(VAR ev : MGMouse.MouseEvent;
 pointerBusy: BOOLEAN);
VAR
  handles: MG.WindowHandleSet;
  i      : Word;
BEGIN
  IF NOT pointerBusy THEN BEGIN
    MG.GetWinHandlesAtPos(ev.scrX,ev.scrY,handles);
    IF handles <> oldHandles THEN BEGIN
      oldHandles:= handles;
      ev.events:= ev.events + winChanged;
      i:= maxWinIdx;
      WHILE (i > 0) AND NOT (winHandle[i] IN handles) DO
        Dec(i);
      ev.user[0]:= i;
      MGMouse.SetSymbolPointer(mousePtr[i]);
    END;
  END;
END; (*WinDetect*)

MGMouse.SetMouseEventProc(WinDetect);
oldHandles:= [];

The example is taken from GAHit.

**GetMouseEventProc**

P GetMouseEventProc(VAR EventProc: MouseEventProc (*O*));

This procedure reads the current mouse event procedure and returns it in the variable parameter EventProc.

SV.Result is always assigned to SV.ok
Mouse Pointer

**SetSymbolPointer**

\[ P \text{SetSymbolPointer}(ptr: \text{SymbolPointer}); \]

This procedure installs a (new) mouse pointer of the symbol type. The pointer's appearance is defined by using the parameter `ptr` (see the data type `SymbolPointer`).

The mouse pointer is drawn using the symbol, drawing mode, foreground and background colors that are specified in `ptr`. The pointer is drawn with the symbol's hot spot (drawing point) at the mouse's current position.

The other elements in the current mouse mode such as the pointer's sensitivity, protection of the mouse pointer, the location and size of the limiting window and the pointer's level of visibility are not changed by the call. If the previous pointer were visible before the call was made, then the new pointer will also be visible. In all other cases, `ShowPointer` must be called.

**MGMouse** defines a number of pre-declared standard pointers ready to be used as parameters to `SetSymbolPointer`. See "Constants" above.

**SV.Result**

- **MGErr.invalidDrawMode** invalid drawing mode
- **MGErr.outOfMem** not enough memory to protect the screen background
- **invalidPointer** invalid symbol
- **MGErr.invalidDeviceColor** invalid device color

as well as other errors from the unit **MG**.
VAR
  pointers            : MG.Symbols;
  pointer1, pointer2: MGMouse.SymbolPointer;
  ..
MGMouse.StartMouseEvents(MG.CurScr);
MGMouse.SetSymbolPointer(MGMouse.vertBar0);
MGMouse.ShowPointer;
 ..
MGMouse.LoadSymbols(pointers,'POINTERS.SBS',FALSE);
WITH pointer1 DO BEGIN
  symb      := pointers;
  symbol    := 3;
  drwMode   := MG.complement;
  foreground:= MG.CurMaxDeviceColor;
  background:= 0;
END;
WITH pointer2 DO BEGIN
  symb      := MG.Symbols(MG.CurFont);
  symbol    := ORD('^');
  drwMode   := MG.stamp;
  foreground:= MG.MatchingForeground(MG.yellow);
  background:= 0;
END;
MGMouse.SetSymbolPointer(pointer1);
 ..
MGMouse.SetSymbolPointer(pointer2);
 ..

GetSymbolPointer

P GetSymbolPointer(VAR ptr: SymbolPointer (*O*));

This procedure reads and returns the current mouse pointer in the variable parameter ptr. The current pointer must be of symbol type (SymbolPointer).

SV.Result
invalidPointer current pointer is not of the symbol type
SetMaskPointer

P SetMaskPointer(ptr: MaskPointer);

This procedure installs a (new) mouse pointer of the masked symbol type. The look of the pointer is defined by the parameter ptr (see the data type MaskPointer).

The mouse pointer is drawn using the symbol, mask symbol, foreground and background colors that are specified in ptr. The pointer is drawn with the symbol's hot spot (drawing point) at the current position of the mouse.

The other elements in the current mouse mode such as the pointer's sensitivity, protection of the mouse pointer, the location and size of the limiting window and the pointer's level of visibility are not changed by the call. If the previous pointer were visible before the call was made, then the new pointer will also be visible. In all other cases, ShowPointer must be called.

MGMouse defines a number of predeclared standard pointers ready to be used as parameters to SetMaskPointer. See "Constants" above.

SV.Result
MGErr.outOfMem not enough memory to protect the screen background
invalidPointer invalid symbol or mask symbol
MGErr.invalidDeviceColor invalid device color

as well as errors from the unit MG.

VAR
  maskPointers: MG.Symbols;
  mousePtr    : MGMouse.MaskPointer;
  ..
MGMouse.StartMouseEvents(MG.CurScr);
(* Show the default pointer *)
MGMouse.ShowPointer;
  ..
(* Use a standard pointer *)
MGMouse.SetMaskPointer(MGMouse.checkMark);
  ..
(* Change the color of a standard pointer *)
mousePtr:= MGMouse.hand;
mousePtr.foreground:= MG.MatchingForeground(MG.lightRed);
MGMouse.SetMaskPointer(mousePtr);
  ..
(* Use your own symbols *)
MG.LoadSymbols(maskPointers,'POINTERS.SBS',FALSE);
WITH mousePtr DO BEGIN
  symb := maskPointers;
  symbol := 3;
  maskSymbol := 128 + 3;
  foreground := MG.CurMaxDeviceColor;
  background := 0;
END;
MGMouse.SetMaskPointer(mousePtr);

GetMaskPointer

P GetMaskPointer(VAR ptr: MaskPointer (*O*));

This procedure reads and returns the current mouse pointer in the variable parameter ptr. The current pointer must be of the mask type (MaskPointer).

SV.Result
invalidPointer current pointer is not of the mask type

SetImagePointer

P SetImagePointer(ptr: ImagePointer);

This procedure installs a (new) mouse pointer of the image type. The pointer's appearance is defined using the parameter ptr (see the data type ImagePointer).

The mouse pointer is drawn using the image, position, pointer size, hot spot (drawing point), drawing mode, foreground and background colors that are specified in ptr. If the pointer image is multi-colored (not monochrome) the pointer is not affected by the device color for the foreground and background color.

The other elements in the current mouse mode such as the pointer's sensitivity, protection of the mouse pointer, the location and size of the limiting window and the pointer's level of visibility are not changed by the call. If the previous pointer were visible before the call was made, then the new pointer will also be visible. In all other cases, ShowPointer must be called.
SV.Result
MGErr.invalidDrawMode invalid drawing mode
MGErr.outOfMem not enough memory to protect
the screen background
MGImg.invalidImage invalid image variable
invalidPointer invalid symbol
MGErr.invalidDeviceColor invalid device color

as well as errors from the unit MG.

VAR
  pointerImg: MGImg.Image;
  mousePtr  : MGMouse.ImagePointer;
 ..
WITH mousePtr DO BEGIN
  img              := pointerImg;
  leftSkip         := 0;
  bottomSkip       := 0;
  width            := 16;
  height           := 16;
  hotX             := 3;
  hotY             := 12;
  drwMode          := MG.stamp;
  foreground       := MG.CurMaxDeviceColor;
  background       := 0;
END;
MGMouse.SetImagePointer(mousePtr);
MGMouse.ShowPointer;
 ..
WITH mousePtr DO BEGIN (* modify parts of mousePtr *)
  leftSkip         := 48;
  bottomSkip       := 32;
  width            := 8;
  height           := 16;
  hotX             := 4;
  hotY             := 8;
  drwMode          := MG.replace;
END;
MGMouse.SetImagePointer(mousePtr);

GetImagePointer

P GetImagePointer(VAR ptr: ImagePointer (*O*));

This procedure reads and returns the current mouse pointer in the variable parameter ptr. The current pointer must be of the image type (ImagePointer).
Ref 30-30

MGMouse - Mouse Support

SV.Result
invalidPointer current pointer is not of the image type

CurPointerType

F CurPointerType: MousePointerType;

This function returns the type of the current mouse pointer (symbol, mask or image).

SV.Result is always assigned to SV.ok.

SetPointerFG

P SetPointerFG(color: DeviceColor);

This procedure sets the device color for the mouse pointer's foreground color if the pointer is of the monochrome type. The call does not change the colors of multi-colored image pointers.

As a default, the device color set when the mouse pointer was installed is used.

SV.Result
MGErr.invalidDeviceColor invalid device color

MGMouse.SetPointerFG(MG.MatchingForeground(MG.lightRed));

CurPointerFG

F CurPointerFG: DeviceColor;

This function reads and returns the device color for the mouse pointer's foreground color.

SV.Result is always assigned to SV.ok.
SetPointerBG

P SetPointerBG(color: DeviceColor);

This procedure sets the device color for the mouse pointer's background color if the pointer is monochrome. The call does not change the colors of multi-colored image pointers.

As a default, the device color set when the mouse pointer was installed is used.

SV.Result
MGErr.invalidDeviceColor invalid device color

MGMouse.SetPointerBG(MG.MatchingBackground(MG.blue));

CurPointerBG

F CurPointerBG: DeviceColor;

This function reads and returns the device color for the mouse pointer's background color.

SV.Result is always assigned to SV.ok.

ShowPointer

P ShowPointer;

This procedure increases the mouse's "visibility" by one step. Inside MGMouse there is a counter which keeps track of the mouse's degree of "visibility". Each call to ShowPointer increases the visibility one step, unless the mouse pointer is already fully visible. Each call to HidePointer decreases the visibility one step. The pointer is shown only when the value "fully visible" has been reached. Calls to HidePointer and ShowPointer are normally paired off with each other. ShowPointer does not necessarily make the pointer visible. It all depends on how many calls to HidePointer that have been made prior to this (see "The Mouse Pointer's Visibility" above).

When the unit MGMouse is initialized, and after the call is made to StartMouseEvents, the mouse is one step away from being fully visible. All that is needed is a single call to ShowPointer to make the pointer visi-
ShowPointer does not use the error handling in SV. Therefore, the result cannot be checked by calling SV.Result after the call is made.

MGMouse.HidePointer;
.. MGMouse.ShowPointer;

**HidePointer**

P HidePointer;

This procedure decreases the mouse's "visibility" one step (see ShowPointer).

HidePointer does not use the error handling in SV. Therefore, the result cannot be checked by calling SV.Result after the call is made.

**HideScanWaitPointer**

P HideScanWaitPointer;

Identisk med HidePointer, men avvaktar att bildsvepet har passerat lägsta punkten hos muspekaren innan denna döljs. Om man direkt efter anropet ritar någon grafik och snabbt gör muspekaren synlig med ShowPointer, är det möjligt att hitta med hela operationen innan bildsvepet ånyo passerar och visar upp förändringen på skärmen. Detta under förutsättning att den ritade grafiken kan ritas mycket snabbt. Vinsten är minskat blinkande.

**PointerVisible**

F PointerVisible: BOOLEAN;

This function returns TRUE if the mouse pointer is fully visible, otherwise FALSE.

PointerVisible does not use the error handling in SV. Therefore, the result cannot be checked by calling SV.Result after the call is made.
SetReducedFlicker

P SetReducedFlicker(onoff: BOOLEAN);

When calling this procedure with the parameter onoff set to TRUE, MGMouse tries to synchronize the mouse pointer tracking with the video controller's scan process. This will, especially if the mouse pointer symbol is large, at least reduce flickering of the mouse pointer. However, the mouse tracking will need more time when it has to wait for the scan process.

Calling SetReducedFlicker has only effect if the graphics device supports MG.ScanWait.

The default mode is TRUE.

SetReducedFlicker does not use the error handling in SV. Therefore, the result cannot be checked by calling SV.Result after the call is made.

SetPointerRedrawTime

P SetPointerRedrawTime(scanTimeParts: Word);

This procedure is used to optimize the mouse pointer tracking when SetReducedFlicker is set to TRUE. scanTimeParts specifies the estimated time to draw the mouse pointer in relation to the scan process time. A value of 1000 denotes that the drawing time equals the scanning time (approximately 1/50 sec to 1/80 sec). If the drawing time >= 1000, then SetReducedFlicker should be set to FALSE.

The optimal value depends on the mouse pointer type and pointer size, as well as the speed of the processor and graphics adapter. Normally, it is not necessary to call this procedure, but you can get some improvements in mouse tracking, if you're running on a speedy computer and a fast graphics adapter.

CurPointerRedrawTime

F CurPointerRedrawTime: Word;

Returns the current estimated time to redraw the mouse pointer.
Protection of the Mouse Pointer

**SetPointerProtection**

```pascal
P SetPointerProtection(onoff: BOOLEAN);
```

This procedure controls if the automatic protection of the mouse pointer when calling drawing routines in *MultiGraphics* is to be *on* (*onoff* = TRUE) or *off* (*onoff* = FALSE).

When the call is made to *StartMouseEvents*, the protection is set to *on*. If the mouse pointer is not protected by *MGMouse*, it must be switched off (*HidePointer*) before each drawing operation that may damage the mouse pointer and be switched on afterwards (*ShowPointer*).

The result is always *SV.ok*.

**SetMouseWinLimits**

```pascal
P SetMouseWinLimits(win: Window);
```

This procedure limits the mouse's freedom of movement to the screen area covered by the window *win*, *the limiting window*. The mouse pointer (or more correctly, the pointer's hot spot) cannot move outside this area from now on.

By calling *StartMouseEvents*, the mouse is allowed to move over the complete screen.

**SV.Result**

- *invalidMouse*  
  - mouse is missing or invalid external mouse driver

- *MGErr.invalidWin*  
  - invalid window
Mouse Sensitivity

SetMouseSens

P SetMouseSens(horPixPerMM, vertPixPerMM: Word);

This procedure sets the mouse's sensitivity using the unit of measurement, screen pixels per mm (millimeter) horizontal mouse movement, and vertical mouse movement respectively. The default values are 11 pixel/mm horizontally and 11 pixels/mm vertically.

SV.Result
invalidMouse mouse missing or invalid driver
invalidSens invalid sensitivity value

MGMouse.SetMouseSens(2,1);

SetDoubleSpeedThreshold

P SetDoubleSpeedThreshold(mmPerSecond: Word);

This procedure specifies the speed in mm/s that the mouse must reach in order to double its sensitivity. This acceleration makes it easier to move the mouse quickly over large distances. The default value for the speed limit is 8 mm/s.

SV.Result
invalidMouse mouse missing or invalid driver

SetDoubleClickInterval

P SetDoubleClickInterval(milliSeconds: Word);

This procedure sets the maximum time that is allowed to elapse between two pressings (releasings actually) of the same mouse button so that these together are to be regarded as a double-click. The default value is set when calling StartMouseEvents to 500 ms, i.e. 1/2 second.

Always returns SV.Result = SV.ok.

MGMouse.SetDoubleClickInterval(1000);
The Mouse's Position and Button Status

**SetMouseScrPos**

P SetMouseScrPos(scrX, scrY: INTEGER);

This procedure sets a new position for the mouse pointer. The position is specified in *screen device coordinates*. If the mouse pointer is visible, the pointer is automatically moved to the new position.

SV.Result
invalidMouse mouse missing or invalid driver
invalidPos position outside the screen

MGMouse.SetMouseScrPos(MG.CurScrWidth DIV 2,
                        MG.CurScrHeight DIV 2);

**SetMouseWinPos**

P SetMouseWinPos(winX, winY: INTEGER);

This procedure sets a new position for the mouse pointer. The position is specified in the current window's *window coordinates*. If the mouse pointer is visible, the pointer is automatically moved to the new position.

SV.Result
MGErr.processWithoutWin the calling process has no window
invalidMouse mouse missing or invalid driver
invalidPos position outside current window

MGMouse.SetMouseWinPos(MG.CurWinWidth DIV 2,
                        MG.CurWinHeight DIV 2);
GetMouseScrInfo

P GetMouseScrInfo(VAR buttons : ButtonSet; (*O*)
    VAR scrX, scrY: INTEGER (*O*));

This procedure returns the current position of the mouse in screen device coordinates (scrX, scrY) as well as the status of the buttons (buttons), i.e. which mouse buttons are being held down. GetMouseScrInfo reads the mouse directly and can therefore give a different result than that obtained when mouse events are read from the event queue.

SV.Result
invalidMouse  mouse missing or invalid mouse driver

MGMouse.GetMouseScrInfo(buttons, scrX, scrY);
IF MGMouse.leftButton IN buttons THEN..

GetMouseWinInfo

P GetMouseWinInfo(VAR buttons : ButtonSet; (*O*)
    VAR winX, winY: INTEGER (*O*));

This procedure returns the current position of the mouse in the current window’s window coordinates (winX, winY) as well as the status of the buttons (buttons), i.e. which mouse buttons are being held down. GetMouseWinInfo reads the mouse directly and can therefore give a different result than that obtained when mouse events are read from the event queue.

SV.Result
MGErr.processWithoutWin  the calling process has no window
invalidMouse  mouse missing or invalid mouse driver

MGMouse.GetMouseWinInfo(buttons, winX, winY);
IF (winX < 10) AND (winY < 100) THEN..

MouseInWin

F MouseInWin(win: Window): BOOLEAN;

This function returns TRUE if the mouse pointer is within the area covered by the window win, otherwise FALSE. MouseInWin reads the mouse directly and can therefore give a different result than that obtained when mouse events are read from the event queue.
It returns `FALSE` if the mouse is missing or not initialized and if the window `win` does not belong to the mouse screen or `win` is an invalid window.

The result code in `SV` is not set. Therefore, the result cannot be checked by calling `SV.Result`.

```pascal
PROCEDURE GetButtonPressInfo(    pressedButton: Button;
VAR buttons : ButtonSet; (*O*)
VAR count : Word; (*O*)
VAR lastWinX,
     lastWinY : INTEGER (*O*));
```

For the given mouse button (`pressedButton`), the procedure returns how many times the button has been pressed since the previous call (`count`), the window position in the current window when the latest button was pressed (`lastWinX, lastWinY`) as well as which buttons were held down when the call was made (`buttons`).

The counter is reset to zero after the call is made.

```
SV.Result
MGErr.processWithoutWin the calling process has no window
invalidMouse mouse missing or invalid mouse driver

MGMouse.GetButtonPressInfo(MGMouse.leftButton,
    buttons, leftCount, leftX, leftY);
IF (leftCount = 2) AND (leftButton IN buttons) THEN ..
```

```pascal
PROCEDURE GetButtonReleaseInfo( releasedButton: Button;
VAR buttons : ButtonSet; (*O*)
VAR count : Word; (*O*)
VAR lastWinX,
     lastWinY : INTEGER (*O*)
```

For the given mouse button (`releasedButton`), the procedure returns how many times the button has been released since the previous call (`count`), the window position in the current window when the latest button was released (`lastWinX, lastWinY`) as well as which buttons were held down when the
call was made (buttons).

The counter is reset to zero after the call is made.

\[ \text{SV.Result} \]
\[ \text{MGErr.processWithoutWin} \]
\[ \text{invalidMouse} \]

the calling process has no window

mouse missing or invalid mouse driver

\[ \text{MGMouse.} \text{GetButtonReleaseInfo} \]
\[ \text{MGMouse.rightButton,} \]
\[ \text{buttons,} \]
\[ \text{rightCount,} \]
\[ \text{rightX,} \text{rightY);} \]

\[ \text{IF} \ (\text{rightCount} = 1) \ \text{AND} \]
\[ (\text{rightX} < 100) \ \text{AND} \ (\text{rightY} < 100) \ \text{THEN} \ .. \]

Mouse Emulation

\[ \text{SetEmulationOnly} \]

\[ \text{P} \ \text{SetEmulationOnly(trueOrFalse: BOOLEAN);} \]

This procedure decides if the mouse emulation should be used for itself or in cooperation with an existing real mouse. If the parameter \text{trueOrFalse} equals \text{FALSE} the real mouse, if present, is examined at the same time. \text{trueOrFalse} = \text{TRUE} will not activate a mouse, if present, and will only respond to emulated events.

\[ \text{The procedure must be called before the mouse handler is activated by StartMouseEvents.} \]

A pure emulated mouse will have three "buttons".

\[ \text{SV.Result} \]
\[ \text{mouseHandlerActive} \]

the mouse handler is already active
EmulateMouseEvent

P EmulateMouseEvent (events : BitSet;
               buttons : ButtonSet;
               scrX, scrY : INTEGER);

This procedure emulates all mouse events by software only. The events are placed in the event queue in the same way as genuine mouse events. At call time, the parameter events containing the event types of the emulated event (see "Constants" above) are specified, in buttons the status of the mouse buttons (which keys are pressed) and in scrX and scrY respectively, the position of the mouse expressed in screen device coordinates. The shift keys status, however, is read from the real keyboard.

If a mouse and a mouse driver is activated the emulated events may be mixed with real events whenever you like. The time of the event is set to the time when the call is made to EmulateMouseEvent.

SV.Result
invalidMouse  mouse handling not activated

MGMouse.EmulateMouseEvent ([MGMouse.mouseMoved,
                              MGMouse.leftButtonDoubleClick],
                             [],
                             100, 100);
The unit **MGBez/MGBez87** contains procedures to find an optimal fitting of a sequence of Bézier curves to an array of points. The technique used is based on a method developed by Philip J. Schneider as documented in "Graphics Gems" (first book) from Academic Press.

Bézier fitting is suitable for manipulations of point data from contour tracings, using **MG.TraceOutline** or **MG.TraceComplexOutline**, for conversions of bitmapped fonts to Bézier fonts, etc.. The Bézier curves can be scaled, rotated, sheared, etc..

When calling the fitting procedures, you will specify the largest allowed pixel deviation between points in the original array and the Bézier curves. With this restriction in mind, the algorithm tries to find the lowest number of Bézier curves. The process is fully automatic!

Even if the fitting process is automatic, it is often recommended to "give it a hand" by removing coincident points from the point array and reducing the number of points resulting from trace operations. By locating "corners", that is points where the tangent of the curve makes a leap, and fit each sub array of points separately, the number of Bézier elements can be reduced.

The algorithm is based on floating point numbers and is very intense in computing resources. Our recommendation is that you should only use it with powerful hardware (fast 386's or better) equipped with a co-processor (80387, 486DX, Pentium) - otherwise the computing times will be very long.
Program Exemple

GATrace and Bezier show Bézier fitting to point arrays.

Variables

```pascal
revision: Word;
EP      : SV.ErrProc;
```

Procedures & functions

```pascal
FitIntPolyBezier

P FitIntPolyBezier(VAR points; {ARRAY OF MG.Point} (*I*)
  nPoints: Word;
  maxErr : Real;
  maxNBez: Word;
  VAR polyBez; {ARRAY OF MG.Bezier} (*O*)
  VAR nBez   : Word (*O*));
```

This procedure fits an optimized sequence of integer based Bézier elements to the point array `points` with `nPoints` integer based points (start index = 0). `maxErr` specifies the maximal accepted distance (> 0.0), expressed in pixels, between the original points and the Bézier curves. A larger `maxErr` value usually results in fewer Bézier elements, but at the same time with a larger deviation from the original.

The Bézier elements are returned in the array `polyBez`, holding up to `maxNBez` Bézier elements. The number of elements is returned in `nBez`. `polyBez` should be declared so that `maxNBez >= nBez`.

SV.Result
MGErr.outOfMem not enough memory to continue fitting
MGErr.tooManyPoints more Bézier elements than `polyBez` can store
### Code Snippet

```pascal
CONST
  maxPoints = 1000;
  maxBez = 100;
VAR
  points : ARRAY[0..maxPoints - 1] OF MG.Point;
  polyBez : ARRAY[0..maxBez - 1] OF MG.Bezier;
  nPoints,nBez: Word;
.. 
MG.TraceOutline(x,y,maxPoints,0,points,nPoints);
MGBez.FitIntPolyBezier(points,nPoints,2.0,
                        maxBez,polyBez,nBez);
MG.TransformPolyBezier(polyBez,0,nBez - 1,
                      2000,2000,450,0,0);
MG.DrawPolyBezier(polyBez,0,nBez - 1);
```

### `FitRealPolyBezier`

```pascal
P FitRealPolyBezier(VAR points; {ARRAY OF MGW.Point} (*I*)
  nPoints: Word;
  maxErr : Real;
  maxNBez: Word;
  VAR polyBez;{ARRAY OF MGW.Bezier} (*O*)
  VAR nBez : Word (*O*));
```

This procedure fits an optimized sequence of floating-point based Bézier elements to the point array `points` with `nPoints` floating-point based points (start index = 0). `maxErr` specifies the maximal accepted distance (> 0.0), expressed in pixels, between the original points and the Bézier curves. A larger `maxErr` value usually results in fewer Bézier elements, but at the same time with a larger deviation from the original.

The Bézier elements are returned in the array `polyBez`, holding up to `maxNBez` Bézier element. The number of elements is returned in `nBez`. `polyBez` should be declared so that `maxNBez >= nBez`.

### SV.Result

- **MGErr.outOfMem** not enough memory to continue fitting
- **MGErr.tooManyPoints** more Bézier elements than `polyBez` can store
Drivers
The graphics driver MGCGA (CGA.MGV, CGA.MGA/CGA.XGA) works with IBM's CGA cards and compatible graphics cards. It can even be used with EGA and VGA cards, although the color palettes may not function properly.

MGCGA has only one mode with a resolution of 640x200 pixels and two simultaneous colors. This is how you activate the CGA mode:

```pascal
MG.LoadDevice(driver,'CGA.MGA'); (*reellt läge *)
MG.SetDevice(driver);
```

Screens

The Visible Screen

Driver: CGA.MGV or CGA.MGA/CGA.XGA

The visible screen comprises the monitor's screen. All drawing modes and drawing operations can be used (1 bit/pixel and 1 color plane). The width is 640 pixels and the height 200 pixels. The monitor screen's lower left corner has x = 0 and y = 0.
Drivers D1-2

Virtual Screens
Driver: CGA.MGA/CGA.XGA

All drawing modes and drawing operations can be used (1 bit/pixel and 1 plane). Copying to and from the visible screen is allowed. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A virtual screen having the same dimensions as the visible screen, i.e. 640x200 pixels requires 16 KB.

Mono Virtual Screens
The same data as for virtual screens.

Special Screens
Not available.

Optimizing Speed when Copying
When copying between screens or within screens, maximum performance is obtained if you remember the following points:

- avoid copying between x positions that force the bit pattern to be shifted. The distance in the x value should be an even multiple of 8

- use the drawing mode replace

The same advice also applies when drawing images. Also try to output text characters at byte boundaries in the x direction, i.e. the x value ought to be a multiple of 8 (fast positions are obtained by using MG.GetFastXY).
Line Style

Line styles on virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type `MGPat.LineStyle16Struc`. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

```pascal
CGALineStyle = ^MGPat.LineStyle16Struc;
```

Generic line styles in `MGPat` can be used.

Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

```pascal
PatternStruc = RECORD
  strucSize: Word; (* = 6 + patSize *)
  byteWidth: Word;
  height   : Word;
  pat      : ARRAY[1..patSize] OF Byte;
END;
CGAPattern = ^PatternStruc;
```

The size of the field `pat` is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Usually 8x8 bits pattern of the internal type `MGPat.Pattern8Struc` are used. See `MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in `MGPat` can be used.
Standard Symbols

MGCGA defines all MultiGraphics standard symbols - see "MG - Symbols". The symbol's size varies, however, the maximum size is 16x16 pixels.

The Default Font

The default font has a fixed width and height of 8x8 pixels. This gives 80 characters per line and 25 lines when the resolution is 640x200 pixels.

PCX Images

MGImg can read and write monochrome PCX images.

Family Data

deviceFamily   6
deviceType    rasterDevice
visibleWidth  640
visibleHeight 200
bitsPerPixel   1
maxColorPlane  0
aspect         420
maxDeviceColor 1
maxColorMask   1
maxPaletteIndex 1
maxPaletteColor 15

Modes

cga640x200   0
maxDeviceMode 0

The driver supports only one mode.
Colors

CGAPaletteColor = Word;

CGAPaletteArray = ARRAY[0..maxPaletteIndex] OF CGAPaletteColor;

CGAPaletteStruc = RECORD
  strucSize: Word;
  data     : CGAPaletteArray;
END;

CGAPalette = ^CGAPaletteStruc;

Max. device color (color index) is 1. The palette color for device color 0 is always black, while the palette color for device color 1 can be chosen from among 16 different colors in accordance with the following scheme:

<table>
<thead>
<tr>
<th>CGAPaletteColor</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>black</td>
</tr>
<tr>
<td>1</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>green</td>
</tr>
<tr>
<td>3</td>
<td>cyan</td>
</tr>
<tr>
<td>4</td>
<td>red</td>
</tr>
<tr>
<td>5</td>
<td>magenta</td>
</tr>
<tr>
<td>6</td>
<td>brown</td>
</tr>
<tr>
<td>7</td>
<td>lightGray</td>
</tr>
<tr>
<td>8</td>
<td>darkGray</td>
</tr>
<tr>
<td>9</td>
<td>lightBlue</td>
</tr>
<tr>
<td>10</td>
<td>lightGreen</td>
</tr>
<tr>
<td>11</td>
<td>lightCyan</td>
</tr>
<tr>
<td>12</td>
<td>lightRed</td>
</tr>
<tr>
<td>13</td>
<td>lightMagenta</td>
</tr>
<tr>
<td>14</td>
<td>yellow</td>
</tr>
<tr>
<td>15</td>
<td>white</td>
</tr>
</tbody>
</table>
The following example sets the palette to black-red:

```pascal
VAR
cgaPalStruc: MGCGA.CGAPaletteStruc;
..
cgaPal.strucSize := SizeOf(cgaPalStruc);
cgaPal.data[0] := 0;
cgaPal.data[1] := 4;
MG.SetDevicePalette(MG.DevicePalette(@cgaPalStruc));
```

The palette can, of course, be set more simply (and be more portable) using `MG.SetRGB`:

```pascal
MG.SetForeground(1); MG.SetRGB(MG.red);
```
The graphics driver **MGMCGA** (MCGA.MGV, MCGA.MGA/MCGA.XGA) works with IBM's MCGA cards and all VGA compatible graphics cards.

**MGMCGA** has only one mode with a resolution of 320x200 pixels and 256 simultaneous colors. This is how you activate the MCGA mode:

```pascal
MG.LoadDevice(driver,'MCGA.MGA'); (*real mode*)
MG.SetDevice(driver);
```

**Screens**

**The Visible Screen**

Driver: MCGA.MGV or MCGA.MGA/MCGA.XGA

The visible screen comprises the monitor's screen. All drawing modes and drawing operations can be used (8 bit/pixel and 1 color plane). The width is 320 pixels and the height 200 pixels. The monitor screen's lower left corner has \( x = 0 \) and \( y = 0 \).

**Virtual Screens**

Driver: MCGA.MGA/MCGA.XGA

All drawing modes and drawing operations can be used (1 byte/pixel,
bits/pixel and 1 plane). Copying to and from the visible screen is allowed. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A virtual screen having the same dimensions as the visible screen, i.e. 320x200 pixels requires 64 KB.

Mono Virtual Screens

1 bit/pixel. Otherwise, the same data as for virtual screens.

Special Screens

Not available.

Line Style

Line styles on virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type MGPat.LineStyle16Struc. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

MCGALineStyle = ^MGPat.LineStyle16Struc;

Generic line styles in MGPat can be used.

Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

PatternStruc = RECORD
  strucSize: Word; (* = 6 + patSize *)
  byteWidth: Word;
  height : Word;
  pat     : ARRAY[1..patSize] OF Byte;
END;

MCGAPattern = ^PatternStruc;
The size of the field `pat` is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Usually 8x8 bits patterns of the internal type `MGPat.Pattern8Struc` are used. See `MGPat - Standard Patterns` for declarations and examples on user defined patterns.

Generic patterns in `MGPat` can be used.

**Standard Symbols**

`MGMCGA` defines all `MultiGraphics` standard symbols - see "MG - Symbols". The symbol's size varies, however, the maximum size is 16x16 pixels.

**The Default Font**

The default font has a fixed width and height of 8x8 pixels. This gives 40 characters per line and 25 lines.

**PCX Images**

`MGImg` can read and write both 256-colors and monochrome images on PCX format. PCX color images for MCGA have the same internal format as PCX images for Super VGA 256 color modes and 8514/A but differ from images for VGA, EGA, Hercules, CGA etc. Also PCX images for EGA and VGA can be used, but they are converted to 256 color format when loaded.
Family data

deviceFamily $13 (* 19 *)
deviceType rasterDevice
bitsPerPixel 8
maxColorPlane 0
maxDeviceColor 255
maxColorMask 255
maxPaletteIndex 255
maxPaletteColor 63

Modes

mrgba320x200x256 0
maxDeviceMode 0

The driver supports only one mode.

visibleWidth : INTEGER; (* 320 *)
visibleHeight: INTEGER; (* 200 *)
aspect : Word; (* 840 *)

Screen width, height and aspect are all variables, but they have constant values because MGMCGA only supports one mode, 320x200 pixels.

Colors

MCGAPaletteColor = RECORD (* 3 bytes *)
  r: Word8;
  g: Word8;
  b: Word8;
END;
MCGAPalleteArray =
   ARRAY[0..maxPaletteIndex] OF MCGAPalleteColor;

MCGAPalleteStruc = RECORD
   strucSize: Word;
   data     : MCGAPalleteArray;
END;

MCGAPallete = ^MCGAPalleteStruc;

MCGA can show 256 simultaneous colors. The color index (device color) lies within the range of 0..255. The palette color for each device color may in turn be set as an RGB value with 64 levels of intensity (0..63) for red, green and blue respectively (data type MCGAPalleteColor). Each palette color may be chosen from among $64^3 = 262144$ colors!

The default palette (MG.SetDefaultPalette) has been chosen so that the first sixteen colors (0..15) corresponds to the predefined RGB colors in the unit MG. These 16 colors have the same color indexes as the same colors for EGA and VGA. The remaining colors (16..255) are selected so that the green intensity changes using small increments, while red and blue change using larger increments. The secret behind this is to utilize the differences in the eye's sensitivity to red, green and blue colors. Both device colors 15 and 255 are set to white.

The following is an example of reading the current palette, how the color index 0 is set to white and color index 255 is set to black:

VAR
   mcgaPal: MGMCGA.MCGAPallete;
  ..
MG.GetDevicePalette(MG.DevicePalette(mcgapal));
WITH mcgaPal^ DO BEGIN
   WITH data[0] DO BEGIN
      r:= 63; g:= 63; b:= 63;
   END;
   WITH data[255] DO BEGIN
      r:= 0; g:= 0; b:= 0;
   END;
END;
MG.SetDevicePalette(MG.DevicePalette(mcgapal));
  ..
MG.DisposeDevicePalette(MG.DevicePalette(mcgapal));

Of course, you could equally well have used the hardware independent palette handling in MultiGraphics.
The graphics driver **MGEGA** (EGA.MGV, EGA.MGA/EGA.XGA) works with EGA compatible graphics cards. It can even be used with the VGA card.

**MGEGA** supports two modes, both with resolutions of 640x350 pixels. The default mode (mode 0) gives 16 simultaneous colors and requires at least 128 KB of memory on the EGA card. This is how you activate the default mode:

```pascal
MG.LoadDevice(driver,'EGA.MGA'); (*real mode*)
MG.SetDevice(driver);
```

The other mode is a monochrome mode and is used solely with monochrome monitors. Only 64 KB of graphics memory is needed in this mode. This is how you activate it:

```pascal
MG.LoadDevice(driver,'EGA.MGA'); (*real mode*)
MG.SetDeviceMode(driver,MGEGA.ega640x350x2);
MG.SetDevice(driver);
```

**Screens**

**The Visible Screen**

Driver: EGA.MGV or EGA.MGA/EGA.XGA

The visible screen comprises the monitor's screen. All drawing modes and drawing operations can be used (1 bit/pixel and max. 4 color planes using...
the default mode, 1 color plane in monochrome mode). The monitor screen's lower left corner has $x = 0$ and $y = 0$.

You can attach a Hercules graphics card and a second monochrome monitor and display graphics on both screens in parallel. The dual monitor combination is perfect for debugging graphics applications.

**Virtual Screens**

Driver: EGA.MGA/EGA.XGA

All drawing modes and drawing operations can be used (1 bit/pixel and max. 4 color planes using the default mode, 1 color plane in monochrome mode). Copying to and from the visible screen is allowed. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A virtual screen having the same dimensions as the default mode's visible screen, i.e. 640x350 pixels, with 4 color planes requires 109 KB.

**Mono Virtual Screens**

The same data as for virtual screens. A mono virtual screen having the same dimensions as the default mode's visible screen requires 28 KB.

**Special Screens**

Driver routines: EGA.MGA/EGA.XGA or EGA.MGV

Only *one* special screen can be created. It has the same data and features as a visible screen except for the screen height. The height of the special screen depends on how much memory the graphics card has:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 KB</td>
<td>59 pixels</td>
</tr>
<tr>
<td>256 KB</td>
<td>469 pixels</td>
</tr>
</tbody>
</table>

As can be seen, the special screen is considerably smaller in height than the visible screen if the card has 128 KB, and considerably larger if it has 256 KB. The width is always 640 pixels and the maximum number of color planes is 4.
There are two advantages in using the special screen as opposed to virtual screens:

- economical with memory - the memory is on the graphics card
- faster graphics operations, especially when copying between visible screens and the special screen.

The increased speed (approx. 4 times faster compared to copying to and from virtual screens) is due to the fact that all four color planes are simultaneously copied *internally within* the EGA card when doing block operations. See also the advice given below about how to optimize speed.

A small special screen can be useful when you want to quickly do animation, or handle icons, etc.

```pascal
VAR
    specScr              : MG.Screen;
    specWidth, specHeight: INTEGER;
    specMask             : MG.DeviceColor;
    ..
MG.CreateSpecScr(specScr,specWidth,specHeight,specMask);
(* Now specWidth = 640, specHeight = 59 or 469, specMask = 15 *)
IF specHeight > 350 THEN MG.SetScr(specScr);..
```

### Optimizing Speed

When copying between screens or within screens, maximum performance is obtained if you remember the following points:

- avoid copying between x positions that force the bit pattern to be shifted. The distance in the x value should be a multiple of 8,
- use the drawing mode **replace**,
- limit the number of color planes that are updated (by using `MG.SetColorMask`).

The same advice also applies when drawing images. Also try to output text characters at byte boundaries in the x direction, i.e. the x value ought to be a multiple of 8 (fast positions are obtained by calling `MG.GetFastXY`).
Line Style

Line styles on virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type \texttt{MGPat.LineStyle16Struc}. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

\texttt{EGALineStyle} = \texttt{^MGPat.LineStyle16Struc};

Generic line styles in \texttt{MGPat} can be used.

Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

\texttt{PatternStruc} = \texttt{RECORD}
  \hspace{1em} \texttt{strucSize: Word; } (* = 6 + patSize * )
  \hspace{1em} \texttt{byteWidth: Word;}
  \hspace{1em} \texttt{height : Word;}
  \hspace{1em} \texttt{pat : ARRAY[1..patSize] OF Byte};
\texttt{END;}

\texttt{EGAPattern} = \texttt{^PatternStruc};

The size of the field \texttt{pat} is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Usually 8x8 bits patterns of the internal type \texttt{MGPat.Pattern8Struc} are used. See MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in \texttt{MGPat} can be used.
Standard Symbols

MGEGA defines all MultiGraphics standard symbols - see "MG - Symbols". The symbol's size varies, however the maximum size is 16x16 pixels.

The Default Font

The default font has a fixed width and height of 8x14 pixels. This gives 80 characters per line and 25 lines.

PCX Images

MGImg can read and write monochrome as well as 2, 4, 8 and 16-colors PCX images. PCX color images for VGA have the same internal format as PCX images for EGA but are different from images for 256-colors MCGA, SuperVGA, 8514/A, etc.

Family Data

```
deviceFamily       $10  (* 16 *)
deviceType         rasterDevice
visibleWidth       640
visibleHeight      350
bitsPerPixel       1
aspect             770
maxPaletteIndex    16
maxPaletteColor    63
```
**Modes**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ega640x350x16</td>
<td>0</td>
</tr>
<tr>
<td>ega640x350x2</td>
<td>1</td>
</tr>
<tr>
<td>maxDeviceMode</td>
<td>1</td>
</tr>
</tbody>
</table>

The default mode is **ega640x350x16**.

```pascal
maxColorPlane : Word;
maxDeviceColor : DeviceColor;
maxColorMask  : DeviceColor;
curMaxPaletteIndex: Word;
```

```pascal
ega640x350x16    ega640x350x2
maxColorPlane    3       0
maxDeviceColor   15      1
maxColorMask     15      1
curMaxPaletteIndex 16     1
```

**Colors**

```pascal
EGAPaletteColor = Byte;

EGAPaletteArray =
  ARRAY[0..maxPaletteIndex] OF EGAPaletteColor;

EGAPaletteStruc = RECORD
  strucSize: Word;
  data     : EGAPaletteArray;
END;

EGAPalette = ^EGAPaletteStruc;
```

In the default mode, EGA has access to 16 simultaneous colors using a device color (color index) in the 0..15 range. Each palette color can be chosen at random from a palette of 64 different, but fixed, nuances of color, numbered from 0..63 (**EGAPaletteColor**).
The EGA palette also contains a seventeenth color using color index 16. This color index is used to set the border color of the monitor screen. The color is normally black. This border cannot be drawn on.

The default palette for EGA (**MG.SetDefaultPalette**) corresponds to the predefined RGB colors in the unit **MG**. The palette looks like this:

<table>
<thead>
<tr>
<th>Device color</th>
<th>Palette color</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>black</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>green</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>cyan</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>red</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>magenta</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>brown</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>lightGray</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>darkGray</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>lightBlue</td>
</tr>
<tr>
<td>10</td>
<td>58</td>
<td>lightGreen</td>
</tr>
<tr>
<td>11</td>
<td>59</td>
<td>lightCyan</td>
</tr>
<tr>
<td>12</td>
<td>60</td>
<td>lightRed</td>
</tr>
<tr>
<td>13</td>
<td>61</td>
<td>lightMagenta</td>
</tr>
<tr>
<td>14</td>
<td>62</td>
<td>yellow</td>
</tr>
<tr>
<td>15</td>
<td>63</td>
<td>white</td>
</tr>
<tr>
<td>(16)</td>
<td></td>
<td>black (border color)</td>
</tr>
</tbody>
</table>

The following code shows us how you can read the current palette, then change the device color 0 to white, the device color 15 to black and the frame color to white.

```
VAR
  egaPal: MGEA.EGAPalette;
 ..
  MG.GetDevicePalette(MG.DevicePalette(egaPal));
  WITH egaPal^ DO BEGIN
    data[0]:= 63;
    data[15]:= 0;
    data[16]:= 1;
  END;
  MG.SetDevicePalette(MG.DevicePalette(egaPal));
 ..
  MG.DisposeDevicePalette(MG.DevicePalette(egaPal));
```
Drivers D3-8

MGEA - EGA

MultiGraphics & Borland Pascal/Turbo Pascal

DATABITEN
The graphics driver **MGVGA** (VGA.MGV, VGA.MGA/VGA.XGA) works with all VGA compatible graphics cards.

**MGVGA** supports IBM's original modes with resolutions of 640x480 pixels as well as the 800x600 pixels VESA mode. The default mode (mode 0) gives 16 *simultaneous* colors, where each color can be chosen from a palette of $64^3 = 262144$ colors! This is how you activate the default mode:

```pascal
MG.LoadDevice(driver,'VGA.MGA'); (*real mode*)
MG.SetDevice(driver);
```

The other original mode is the monochrome mode and is activated by the following:

```pascal
MG.LoadDevice(driver,'VGA.MGA'); (*real mode*)
MG.SetDeviceMode(driver,MGVGA.VGA640x480x2);
MG.SetDevice(driver);
```

**MGVGA** also supports all VESA compatible adapters and has direct support for some SuperVGA adapters in 16 colors modes, with resolutions up to 800x600 pixels:

```
* VESA mode 258: 800x600x16
* VideoSeven: 752x410x16, 720x540x16 and 800x600x16.
* Tseng Lab based adapters: 800x600x16.
```
The SuperVGA modes have 16 simultaneous colors. The 800x600 mode on VESA compatible adapters is activated using the following:

MG.LoadDevice(driver,'VGA.MGA'); (*real mode*)
MG.SetDeviceMode(driver,MGVGA.vesa800x600x16);
IF SV.Result = SV.ok THEN BEGIN
  MG.SetDevice(driver);
  ..
ELSE ..

We have included an error test after SetDeviceMode in case the computer has no VideoSeven card.

Screens

The Visible Screen

Driver: VGA.MGV or VGA.MGA/VGA.XGA

The visible screen comprises the monitor's screen. All drawing modes and drawing operations can be used (1 bit/pixel and max. 4 color planes using the default mode, 1 color plane in monochrome mode). The width and height for the default mode is 640x480 pixels. The monitor screen's lower left corner has x = 0 and y = 0.

You can attach a Hercules graphics card and a second monochrome monitor and display graphics on both screen in parallel. The dual monitor combination is perfect for debugging graphics applications.

Virtual Screens

Driver: VGA.MGA/VGA.XGA

All drawing modes and drawing operations can be used (1 bit/pixel and max. 4 color planes using the default mode, 1 color plane in monochrome mode). Copying to and from the visible screen is allowed. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A virtual screen having the same dimensions as the default mode's visible screen, i.e. 640x480 pixels, with 4 color planes
Mono virtual screens

The same data as for virtual screens. A mono virtual screen having the same dimensions as the default mode's visible screen requires 38 KB.

Special Screens

Driver routines: VGA.MGA/VGA.XGA or VGA.MGV

Only one special screen can be created. It has the same data and features as a visible screen except for the screen height. The height of the special screen for the default mode is 339 pixels. The width is always 640 pixels and the maximum number of color planes is 4.

There are two advantages in using the special screen as opposed to virtual screens:

- economical with memory - the memory is on the graphics card
- faster graphics operations, especially when copying between visible screens and the special screen.

The increased speed (approx. 4 times faster compared to copying to and from virtual screens) is due to the fact that all four color planes are simultaneously copied internally within the VGA card when doing block operations. See also the advice given below about how to optimize speed.

A small special screen can be useful when you want to quickly do animation, or handling of icons, etc.

VAR
  specScr       : MG.Screen;
  specWidth, specHeight: INTEGER;
  specMask      : MG.DeviceColor;

MG.CreateSpecScr(specScr,specWidth,specHeight,specMask);
(* Now specWidth = 640, specHeight = 339,
  specMask = 15 *)
MG.SetScr(specScr);..
Optimizing Speed

When copying between screens or within screens maximum performance is obtained if remember the following points:

- avoid copying between $x$ positions that force the bit pattern to be shifted. The distance in the $x$ value should be an multiple of 8 pixels,

- use the drawing mode replace,

- limit the number of color planes that are updated (by using MG.SetColorMask).

The same advice also applies when drawing images. Also try to output text characters at byte boundaries in the $x$ direction, i.e. the $x$ value ought to be a multiple of 8 (fast positions are obtained by calling MG.GetFastXY).

Line Style

Line styles are represented internally as 16-bits values (2 bytes) according to the data type MGPat.LineStyle16Struc. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

VGALineStyle = ^MGPat.LineStyle16Struc;

Generic line styles in MGPat can be used.

Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

PatternStruc = RECORD
    strucSize: Word; (* = 6 + patSize *)
    byteWidth: Word;
    height   : Word;
    pat      : ARRAY[1..patSize] OF Byte;
END;
VGAPattern = ^PatternStruc;

The size of the field pat is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often 8x8 bits patterns of the internal type MGPat.Pattern8Struc are used. See MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in MGPat can be used.

Standard Symbols MGVGA defines all MultiGraphics standard symbols - see "MG - Symbols". The symbols size varies, however, the maximum size is 16x16 pixels.

The Default Font

The default font has a fixed width and height of 8x16 pixels. This gives 80 characters per line and 30 lines on the visible screen in 640x480 resolution.

PCX Images

MGImg can read and write monochrome as well as 2, 4, 8 and 16-colors PCX images. PCX color images for VGA have the same internal format as PCX images for EGA but are different from images for 256-colors MCGA, SuperVGA, 8514/A, etc.

Family Data

deviceFamily $12 (* 18 *)
deviceType rasterDevice
bitsPerPixel 1
maxPaletteIndex 16
maxPaletteColor 63
**Drivers D4-6**  

**Modes**

- `vga640x480x16` 0
- `vga640x480x2` 1
- `videoSeven752x410x16` 2
- `videoSeven720x540x16` 3
- `videoSeven800x600x16` 4
- `tseng800x600x16` 5
- `vesa800x600x16` 258
- `maxDeviceMode` 258

The default mode is `vga640x480x16`.

**Variables**

- `visibleWidth` : INTEGER;
- `visibleHeight` : INTEGER;
- `aspect` : Word;

Data for VGA modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>width</th>
<th>height</th>
<th>aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>vga640x480x16 (default)</td>
<td>640</td>
<td>480</td>
<td>1000</td>
</tr>
<tr>
<td>vga640x480x2</td>
<td>640</td>
<td>480</td>
<td>1000</td>
</tr>
<tr>
<td>videoSeven752x410x16</td>
<td>752</td>
<td>410</td>
<td>800</td>
</tr>
<tr>
<td>videoSeven720x540x16</td>
<td>720</td>
<td>540</td>
<td>1150</td>
</tr>
<tr>
<td>videoSeven800x600x16</td>
<td>800</td>
<td>600</td>
<td>1000</td>
</tr>
<tr>
<td>tseng800x600x16</td>
<td>800</td>
<td>600</td>
<td>1000</td>
</tr>
<tr>
<td>vesa800x600x16</td>
<td>800</td>
<td>600</td>
<td>1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>max. col.planes</th>
<th>max. dev.col</th>
<th>max. col.mask</th>
<th>max. palindx</th>
</tr>
</thead>
<tbody>
<tr>
<td>vga640x480x16</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>vga640x480x2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>videoSeven752x410x16</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>videoSeven720x540x16</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>videoSeven800x600x16</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>tseng800x600x16</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>vesa800x600x16</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>
Colors

maxColorPlane : Word;
maxDeviceColor : MG.DeviceColor;
maxColorMask : MG.DeviceColor;
curMaxPaletteIndex: MG.DeviceColor;

VGAPaletteColor = RECORD
  r: Byte;
  g: Byte;
  b: Byte;
END;

VGAPaletteArray = ARRAY[0..maxPaletteIndex] OF VGAPaletteColor;

VGAPaletteStruc = RECORD
  strucSize: Word;
  data     : VGAPaletteArray;
END;

VGAPalette = ^VGAPaletteStruc;

VGA color modes (including SuperVGA modes) can show 16 colors simultaneously. The device colors (color index) are within the range of 0..15. The palette color for each device color can in turn be set as an RGB value using the 64 intensity levels (0..63) for red, green and blue respectively (using the data type VGAPaletteColor). Each palette color can therefore be chosen from among \(64^3 = 262144\) colors!

The VGA palette also contains a seventeenth color using color index 16. This color index is used to set the border color of the monitor screen. The color is normally black. This border can not be draw on.

The default palette for VGA (MG.SetDefaultPalette) corresponds to the predefined RGB colors in the unit MG. The palette looks like this:
### Device color | Palette color | Color
---|---|---
0 | 0 | black
1 | 1 | blue
2 | 2 | green
3 | 3 | cyan
4 | 4 | red
5 | 5 | magenta
6 | 20 | brown
7 | 7 | lightGray
8 | 56 | darkGray
9 | 57 | lightBlue
10 | 58 | lightGreen
11 | 59 | lightCyan
12 | 60 | lightRed
13 | 61 | lightMagenta
14 | 62 | yellow
15 | 63 | white
.. | .. | ..
(16) | 0 | black (border color)

The following code shows us how we can read the current palette, then change the device color 0 to white, the device color 15 to black and the frame color to white.

```
VAR
  vgaPal: MGVGA.VGAPalette;
  ..
MG.GetDevicePalette(MG.DevicePalette(vgaPal));
WITH vgaPal^ DO BEGIN
  WITH data[0] DO BEGIN
    r:= 63; g:= 63; b:= 63;
  END;
  WITH data[15] DO BEGIN
    r:= 0; g:= 0; b:= 0;
  END;
END;
MG.SetDevicePalette(MG.DevicePalette(vgaPal));
  ..
MG.DisposeDevicePalette(MG.DevicePalette(vgaPal));
```

You can, of course, use the hardware independent palette handling in **MultiGraphics**.
The graphics driver MGHer (HER.MGV, HER.MGA/HER.XGA) works with Hercules and compatible graphics card. MGHer has only one mode with a resolution of 720x348 pixels and the colors black and "light" (the exact color depends on the type of monitor). This is how you activate the Hercules mode:

MG.LoadDevice(driver,'HER.MGA'); (*real mode*)
MG.SetDevice(driver);

Screens

The Visible Screen

Driver: HER.MGV or HER.MGA/HER.XGA

The visible screen comprises the monitor's screen. All drawing modes and drawing operations can be used (1 bit/pixel and 1 color plane). The width is 720 pixels and the height 348 pixels. The monitor screen's lower left corner has x = 0, y = 0. All drawing operations and drawing modes are allowed.

You can attach a CGA, EGA, VGA, SuperVGA or 8514/A graphic adapters and a second color monitor to display graphics on both screen in parallel. The dual monitor combination is perfect for debugging graphics applications.
Virtual Screens

Driver: HER.MGA/HER.XGA

All drawing modes and drawing operations can be used (1 bit/pixel and 1 color plane). Copying to and from visible screens is allowed. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A virtual screen having the same dimensions as the visible screen, i.e. 720x348 pixels, requires 31 KB.

Mono Virtual Screens

The same data as for virtual screens.

Special Screens

Driver routines: HER.MGA/HER.XGA or HER.MGV

Only one special screen can be created. It has the same data and features as the visible screen.

One advantage of the special screen is that it saves memory compared to virtual screens.

Optimizing Speed when Copying

When copying between screens or within screens, the maximum performance is obtained if you remember the following points:

· avoid copying between x positions that force the bit pattern to be shifted. The distance in the x value should be an even multiple of 8 pixels,

· use the drawing mode replace.

The same advice also applies when drawing images. Also try to output text characters at byte boundaries in the x direction, i.e. the x value ought to be a multiple of 8 (fast positions are obtained by calling MG.GetFastXY).
Line Style

Line styles on virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type \texttt{MGPat.LineStyle16Struc}. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

\[
\text{HerLineStyle} = \ ^\text{^MGPat.LineStyle16Struc};
\]

Generic line styles in \texttt{MGPat} can be used.

Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

\[
\begin{array}{l}
\text{PatternStruc} = \text{RECORD} \\
\quad \text{strucSize: Word; (} * = 6 + \text{patSize} * \text{)} \\
\quad \text{byteWidth: Word;} \\
\quad \text{height : Word;} \\
\quad \text{pat : ARRAY[1..patSize] OF Byte;} \\
\end{array}
\]

\[
\text{HerPattern} = \ ^\text{^PatternStruc};
\]

The size of the field \texttt{pat} is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Usually 8x8 bits patterns of the internal type \texttt{MGPat.Pattern8Struc} are used. See MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in \texttt{MGPat} can be used.
Standard Symbols

MGIH - Hercules defines all MultiGraphics standard symbols - see "MG - Symbols". The symbol's size varies, however, the maximum size is 16x16 pixels.

Default Font

The default font has a constant width and height of 8x14 pixels. This gives 90 characters per line and 25 lines.

PCX Images

MGIImg can read and write monochrome images on PCX formats.

Family Data

deviceFamily 7
deviceType rasterDevice
visibleWidth 720
visibleHeight 348
bitsPerPixel 1
maxColorPlane 0
aspect 690
maxDeviceColor 1
maxColorMask 1
maxPaletteIndex 1
maxPaletteColor 1
Colors

HerPaletteColor = Word;

HerPaletteArray =
  ARRAY[0..maxPaletteIndex] OF HerPaletteColor;

HerPaletteStruc = RECORD
  strucSize: Word;
  data     : HerPaletteArray;
END;

HerPalette = ^HerPaletteStruc;

Hercules graphics are monochrome, using the two device colors 0 and 1 and two palette colors 0 and 1.
The graphics driver **MG8514A** (8514A.MGV, 8514A.MGA/8514A.XGA) works with IBM's 8514/A graphics adapter and all compatible adapters. Also IBM XGA supports the 8514/A standard.

**MG8514A** has a default mode which, with IBM's original adapter and expanded graphics memory (1 MB), gives a resolution of 1024x768 pixels and 256 simultaneous colors. The original 8514/A also has support for 640x480 pixels with 256 colors as well as 1024x768 pixels with 64 grey scales.

The 8514/A has on-card hardware support for many graphics primitives which give fast and high resolution graphics and color capabilities. **MG8514A** uses the hardware clipping and the support for multi-tasking found on the 8514/A, which provides excellent performance even when running multi-tasking applications.

The 8514/A can be used side by side with VGA, SuperVGA or Hercules adapters in order to operate two monitor screens. With **MultiGraphics** both screens can be active at the same time. A dual screen system is a great help when debugging graphics applications. It is even possible to run three monitor screens at the same time - 8514/A, a VGA or SuperVGA and a Hercules.
MG8514A is based on the IBM Adapter Interface. The Adapter Interface is incorporated in a memory resident program that must be loaded before activating the graphics applications. Using IBM's original 8514/A, the Adapter Interface is loaded using the command:

```plaintext
HDILOAD <Enter>
```

Here's how the 8514/A is activated from an application program:

```plaintext
MG.LoadDevice(driver,'8514A.MGA'); (*real mode*)
MG.SetDevice(driver);
```

If the Adapter Interface has not been loaded, the error message MGErr.invalidConfig is obtained.

Other modes, e.g. 640x480 pixels and 256 colors, can be activated as follows:

```plaintext
MG.LoadDevice(driver,'8514A.MGA'); (*real mode*)
MG.SetDeviceMode(driver,MG8514A.a8514x640x480);
MG.SetDevice(driver);
```

Screens

The Visible Screen

Driver: 8514A.MGV or 8514A.MGA/8514A.XGA

The visible screen comprises the monitor screen. All drawing modes and drawing operations can be used. MG8514A uses 1 byte/pixel (8 bits/pixel) and 1 color plane. The default mode of the original 8514/A has a width and height of 1024x768 pixels. The monitor screen's lower left corner has x = 0 and y = 0.

Mode 1 (a8514x640x480), of the original 8514/A, has a width and height of 640x480 pixels.

Virtual Screens

Driver: 8514A.MGA/8514A.XGA

All drawing modes and drawing operations can be used (1 byte/pixel (8
bits/pixel) and 1 color plane). Copying to and from the visible screen is allowed. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A virtual screen that has the same size as the visible screen, i.e. 1024x768 pixels, requires 768 KB. We recommend using extended XMS or expanded EMS memory for virtual screens.

Mono Virtual Screens

1 bit/pixel. Otherwise, the same data as for virtual screens. A mono virtual screen having the same dimensions as a visible screen (1024x768) requires 96 KB.

Special Screens

Drivers: 8514A.MGA/8514A.XGA, 8514A.MGV

Only one special screen can be created. It has the same data and features as the visible screen except for the screen height. The special screen's height in default mode is 256 pixels. The width is 1024 pixels with 1 byte/pixel.

There are two advantages in using special screens as opposed to virtual screens:

· economical with memory - the memory is on the graphics card,
· faster graphics operations, especially when copying between visible screens and special screens.

The greater increase in speed (in comparison with copying to and from virtual screens) is due to the fact that copying is taken care of by the hardware in the 8514/A. See also the advice about optimizing speed below.

The memory on the graphics card which contains the special screen is, however, used for some internal graphics operations in the 8514/A. According to IBM's documentation, this applies when doing pattern filling. Avoid therefore FillPolygon, FillShortPolygon, FillCircle, FillEllipse, FillSector and FillEllipseSector when using special screens!

A small special screen can be useful for speedy animations, handling of icons, etc.
VAR
  specScr    : MG.Screen;
  specWidth, specHeight: INTEGER;
  specMask   : MG.DeviceColor;
  ...
MG.CreateSpecScr(specScr, specWidth, specHeight, specMask);
(* Now specWidth = 1024, specHeight = 128, specMask = 255 *)
MG.SetScr(specScr);...

Optimizing Speed

If you follow these recommendations you will get the most out of your 8514/A adapter:

· use hardware clipping (see below),

· use the special screen as a buffer as much as possible (except when doing filling routines)

· use `MG.DrawPolyline` when drawing lines repeatedly,

· avoid the drawing mode, `stamp`, when copying within the visible screen or between the visible screen and a virtual screen or the special screen,

· use fast positions (`MG.GetFastXY`) when working with blocks within mono virtual screens.

Line Style

Line styles are represented internally as 16-bits values (2 bytes) according to the data type `MGPat.LineStyle16Struc`. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

\[A8514ALineStyle = ^{\text{MGPat}}.\text{LineStyle16Struc};\]

Generic line styles in `MGPat` can be used.
Pattern

Patterns are represented internally as rectangular bit patterns where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

```
PatternStruc = RECORD
  strucSize: Word; (*  = 6 + patSize *)
  byteWidth: Word;
  height   : Word;
  pat      : ARRAY[1..patSize] OF Byte;
END;
```

```
A8514APattern = ^PatternStruc;
```

The size of the field pat is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often 16x16 bits patterns of the internal type MGPat.Pattern16Struc are used. See MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in MGPat can be used.

Standard Symbols

MG8514A defines all MultiGraphics standard symbols - see "MG - Symbols". The symbol's size varies, however, the maximum size is 24x24 pixels.

The Default Font

The default font has a constant width and height of 12x24 pixels. This gives 85 characters per line and 32 lines when using a resolution of 1024x768 pixels.
PCX Images

MGImg can read and write both 256-colors and monochrome images on PCX format. PCX color images for 8514/A have the same internal format (byte oriented) as PCX images for MCGA and SuperVGA 256 color, but differ from images for VGA, EGA, CGA etc. Also PCX images for EGA and VGA can be used, but they are converted to 256-colors format when loaded.

Family Data

<table>
<thead>
<tr>
<th>deviceFamily</th>
<th>8514</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviceType</td>
<td>rasterDevice</td>
</tr>
<tr>
<td>bitsPerPixel</td>
<td>8</td>
</tr>
<tr>
<td>maxColorPlane</td>
<td>0</td>
</tr>
<tr>
<td>maxPalette4Index</td>
<td>15</td>
</tr>
<tr>
<td>maxPalette6Index</td>
<td>63</td>
</tr>
<tr>
<td>maxPaletteIndex</td>
<td>255</td>
</tr>
<tr>
<td>maxPaletteColor</td>
<td>255</td>
</tr>
</tbody>
</table>

Modes

| a8514x1024x768 | 0  |
| a8514x640x480  | 1  |
| maxDeviceMode  | 255|

The modes have been numbered in accordance with the original 8514/A card from IBM. Mode 0 gives 1024x768 pixels with 256 colors provided that you have a fully equipped adapter memory and a color monitor, 16 colors if the memory is not expanded, and 64 grey scale levels using a monochrome screen. Mode 1 gives 640x480 pixels with 256 colors if a color monitor is used and 64 grey scales if using a monochrome screen.

The number of available modes and the precise meaning and use of these is specific for each card.
Variables

```pascal
visibleWidth      : INTEGER;
visibleHeight     : INTEGER;
aspect            : INTEGER;
```

The aspect for mode 0 and mode 1 on the original 8514/A is 1000, i.e. 1:1.

Colors

```pascal
maxDeviceColor    : DeviceColor;  (* 15/63/255 *)
maxColorMask      : DeviceColor;  (* 15/63/255 *)
curMaxPaletteIndex: Word;         (* 15/63/255 *)

A8514PaletteColor = RECORD    (* 4 bytes *)
  r  : Byte;
  b  : Byte;
  g  : Byte;
  res: Byte; (* reserved *)
END;

A8514Palette4Array =
  ARRAY[0..maxPalette4Index] OF A8514PaletteColor;

A8514Palette4Struc = RECORD
  strucSize: Word;
  data4    : A8514Palette4Array;
END;

A8514Palette6Array =
  ARRAY[0..maxPalette6Index] OF A8514PaletteColor;

A8514Palette6Struc = RECORD
  strucSize: Word;
  data6    : A8514Palette6Array;
END;

A8514PaletteArray =
  ARRAY[0..maxPaletteIndex] OF A8514PaletteColor;
```
A8514PaletteStruc = RECORD
  strucSize: Word;
  CASE INTEGER OF
    4: (data4 : A8514Palette4Array);
    6: (data6 : A8514Palette6Array);
    8: (data : A8514PaletteArray);
  END;
END;

A8514Palette = ^A8514PaletteStruc;

The original 8514/A with 768 KB can show 256 colors at the same time. The color index (device color) lies within the range of 0..255 (data type A8514PaletteArray). The palette color for each color index can in turn be set as an RGB value using levels of intensity within the range of 0..255 (the original 8514/A uses only the highest 6 bits) for red, green and blue respectively (data type A8514PaletteColor). With the original 8514/A each palette color can be chosen from among $64^3 = 262144$ colors!

The default palette (MG.SetDefaultPalette) has been chosen so that the first sixteen colors (0..15) corresponds to the predefined RGB colors in the unit MG. These 16 colors have the same color indexes as the same colors for EGA and VGA. The remaining colors (16..255) are selected so that the green intensity changes using small increments, while red and blue change using larger increments. The secret behind this is to utilize the differences in the eye's sensitivity to red, green and blue colors. Both device colors 15 and 255 are set to white.

The following is an example of reading the current palette, how the color index 0 is set to white and color index 255 is set to black:

VAR
  a8514Pal: MG8514A.A8514Palette;
  ...
  MG.GetDevicePalette(MG.DevicePalette(a8514Pal));
  WITH a8514Pal^ DO BEGIN
    WITH data[0] DO BEGIN
      r:= 255; g:= 255; b:= 255;
    END;
    WITH data[255] DO BEGIN
      r:= 0; g:= 0; b:= 0;
    END;
  END;
  MG.SetDevicePalette(MG.DevicePalette(a8514Pal));
  ...
  MG.DisposeDevicePalette(MG.DevicePalette(a8514Pal));
Of course, you could equally well have used the hardware independent palette handling in MultiGraphics.

Moreover, palettes are declared with 16 (\texttt{A8514Palette4Array}) and 64 colors (\texttt{A8514Palette6Array}) respectively which are used if the 8514/A card does not have full memory or a monochrome screen (64 grey scales).

\section*{SetDeviceCall Functions}

\subsection*{Clipping - setHardwareClip}

\textbf{Constants}

\begin{itemize}
  \item \texttt{setHardwareClip} = 20
  \item \texttt{hardwareClipOff} = 0
  \item \texttt{hardwareClipOn} = 1
\end{itemize}

Default value: \texttt{hardwareClipOn}.

The default is that the hardware clipping is used on the 8514/A. However, hardware clipping on the original 8514/A only works for graphics that are drawn within the following range of coordinates:

\begin{align*}
  -512 & \leq x \leq 1535 \\
  -512 & \leq y \leq 1535
\end{align*}

If graphics are drawn outside this area, the clipping must be done by software in MultiGraphics. The function \texttt{setHardwareClip} is used to chose the clipping mode by calling \texttt{MG.SetDeviceCall}. Here's how the hardware clipping is switched \textit{off}:

\begin{verbatim}
MG.SetDeviceCall(driver,
  MG8514A.setHardwareClip,
  MG8514A.hardwareClipOff);
\end{verbatim}
The graphics driver MGEps (EPS.MGV, EPS.MGA/EPS.XGA) works with Epson or IBM compatible dot matrix printers as well as some NEC printers. The driver has support for both 9 pin and 24 pin printers, as well as monochrome and color printers. A large number of dot matrix printers (and some laser printers) from other manufacturers can emulate one or more of the modes that are supported.

MGEps can use portrait and landscape orientations as well as several different paper formats, resolutions, fonts, etc.. The default installation has been chosen to also work with older printer models. N.B. that printing routines and printer modes must be selected before activating the printer by MG.SetDevice:

MG.LoadDevice(pr,'EPS.MGA'); (*real mode*)
MG.SetDeviceOutput (pr,MGOut.LPT1);
MG.SetDeviceMode (pr,MGEps...);
MG.SetOrientation (pr,MG..);
MG.SetDeviceCall (pr,MG..,MGEps..);
.. 
MG.SetDevice (pr);
MG.SetBlackOnWhite (TRUE).. 

In the example above, the printout is done via LPT1 (the parallel port). Remember that the printer is a "black-on-white" device!
Screen

The Visible Screen

Driver: EPS.MGV or EPS.MGA/EPS.XGA

The visible screen comprises the paper in the printer. If the orientation is portrait, the paper's lower left corner has the position x = 0, y = 0. If the orientation is landscape the paper's upper left corner has the position x = 0, y = 0. Default is portrait mode.

The only drawing operation permitted is copying from virtual and mono virtual screens respectively to the visible screen. This means that all graphics must be drawn on virtual or mono virtual screens and then afterwards be printed out by copying to the visible screen with MG.CopyView or MG.CopyRect. Since images (Image) are implemented using virtual screens, the graphics can also be printed directly from images with MGImg.DrawImage or MGImg.DrawImagePart.

The only drawing mode permitted is replace and positioning when copying can in general only be done by using decreasing y values (portrait orientation) or increasing x values (landscape orientation), since few printers can feed the paper backwards (NEC is an exception). It is therefore advisable to plan several consecutive copying operations so that you start by copying the virtual screen that is to be placed at the top of the sheet of paper, and so on. There is, of course, nothing to prevent you (except greater memory requirements) from creating a virtual or mono virtual screen that contains the complete sheet of paper and then printing the page in one go.

Before activating the printer by calling MG.SetDevice, a printout procedure must be installed. This is done by calling MG.SetDeviceOutput.

MG.FlushVisiScr is used to advance to a new page.

Virtual and mono virtual screens can be in different graphics devices, provided that these have the same screen structure (bits/pixel, etc). In such cases, you can load EPS.MGV and save some memory.

Virtual Screens

Driver: EPS.MGA/EPS.XGA
All drawing modes and drawing operations can be used (1 bit/pixel, 1 plane in monochrome modes, up to 3 planes when using color modes). Copying to the visible screen is allowed, but not copying from visible screen. Copying to and from virtual screens and mono virtual screens can also be done.

Moreover, printouts using printer specific device fonts are allowed. The internal character buffer contains a maximum of approximately 2500 characters of text which can be printed out with this font type.

Memory requirements: with the highest resolution (180 dots/inch) and a full A4 page virtual screen, you will need approximately 1.1 MB of memory. It is therefore recommended to program in protected mode. In real mode it is often necessary to place virtual screens in expanded EMS, extended XMS or on disk. You can do this by including AutoVM in the USES list.

Mono Virtual Screens

All drawing modes and drawing operations can be used. Copying to the visible screen is allowed, but not copying from visible screen. Copying among virtual screens and mono virtual screens can also be done.

Moreover, printouts using printer device fonts are allowed. The internal character buffer contains a maximum of approximately 2500 characters of text which can be printed out with this font type.

Memory requirements: Approximately 380 KB when printing a full page using 180 dots/inch. Approximately 42 KB when printing a full page using 60 dots/inch.

Special Screens

Not available.

Line Style

Line styles in virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type MGPat.LineStyle16Struc. See "MGPat - Standard Patterns" for examples of user defined line styles.
Drivers D7-4

MGEps - Epson, IBM and NEC printers

EpsonLineStyle = ^MGPat.LineStyle16Struc;

Bits set to one are drawn using the current foreground color while those set to zero are drawn with the background color.

Generic line styles in MGPat can be used.

Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

PatternStruc = RECORD
    strucSize: Word; (* = 6 + patSize *)
    byteWidth: Word;
    height   : Word;
    pat      : ARRAY[1..patSize] OF Byte;
END;

EpsonPattern = ^PatternStruc;

The size of the field pat is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often we use 8x8 bit patterns of type MGPat.Pattern8Struc at lower resolutions and 16x16 patterns of type MGPat.Pattern16Struc at the higher resolutions. See "MGPat - Standard Patterns" for declarations and examples of user defined patterns.

Generic patterns in MGPat can be used.
Standard Symbols

\texttt{MGEps} defines the standard symbols 0..6 in accordance with the description in "MG - Symbols". The symbols do not have masks. The symbol size is specific for each symbol, but the maximum size is 24x24 pixels.

Printer Fonts

\texttt{MGEps} can be used with bitmapped, stroke, outline and Bézier fonts. Moreover \texttt{MGEps} has an hardware supported printer font (of the type \texttt{deviceSymbols}). This font is the device's default font, i.e. the font that can be set with \texttt{MG.SetDefaultFont}. The default font can be varied with regard to typeface, font style, weight and size. This is done by calling \texttt{MG.SetDeviceCall}. All printer modes in \texttt{MGEps} support the font \texttt{MGEps.draft}.

Default fonts and other typefaces: see the table below.

The number of characters that can be printed using the default font is limited to approx. 2500 per screen.

PCX Images

The \texttt{MGImg} unit can both read and write monochrome and 2-, 4-, and 8-colors PCX images. Color PCX images for Epson have the same internal format as color PCX images for EGA and VGA but they do only use the information for a maximum of three color planes instead of four.

Program Outlines

The outlines below show how to draw with full resolution on the printer. In "MG - Copy Graphics" you will also find an example of how to make a hard copy of the monitor screen.

Monochrome Graphics

\texttt{USES MG,MGEps,AutoVM;}

\texttt{VAR}
\begin{verbatim}
  pr   : MG.Device;
  prScr: MG.Screen;
\end{verbatim}
Drivers D7-6

MGEps - Epson, IBM and NEC printers

```pascal
prWin: MG.Window;
..
MG.LoadDevice(pr,'EPS.MGA'); (*real mode*)
MG.SetDeviceOutput (pr,MGOut.LPT1);

(* Select printer modes etc *)
MG.SetDeviceMode (pr,MGEps...);
MG.SetOrientation (pr,MG...);
MG.SetDeviceCall (pr,MG...,MGEps...);
..
MG.SetDevice (pr);

MG.CreateMonoVirtScr (prScr, 
   MG.CurScrWidth, 
   MG.CurScrHeight);

MG.SetScr (prScr);

(* Paper is white so we need to use black-on-white mode *)
MG.SetBlackOnWhite (TRUE);
MG.MatchForeground (MG.black);
MG.MatchBackground (MG.white);
ClrWin; (* Fill screen with white *)

(* Draw graphics *)
..
(* Prepare print-out *)
prWin:= MG.CurWin;
MG.SetVisiScr;

(* Set position on the paper for the lower left corner 
of the graphics *)
MG.SetPos(x,y);

(* Now print *)
MG.CopyView (prWin);

(* New page *)
MG.FlushVisiScr;
```

Color Graphics

USES MG,MGEps,AutoVM;

VAR
   pr   : MG.Device;
   prScr: MG.Screen;
   prWin: MG.Window;
..
MG.LoadDevice(pr,'EPS.MGA'); (*real mode*)
MG.SetDeviceOutput(pr,MGOut.LPT1);

(* Select printer modes etc *)
MG.SetDeviceMode(pr,MGEps.epsLQ25); (* color printer *)
MG.SetOrientation(pr, MG...);
MG.SetDeviceCall(pr, MG..., MGEps...);
MG.SetDevice(pr);
...

MG.CreateVirtScr(prScr,  
   MG.CurScrWidth,  
   MG.CurScrHeight,  
   MG.CurMaxColorMask);

MG.SetScr(prScr);

(* Paper is white so we need to use black-on-white mode *)
MG.SetBlackOnWhite(TRUE);
FillView; (* Fill screen with white *)
MG.MatchForeground(MG.red);
MG.MatchBackground(MG.white);

(* Draw graphics *)
...
(* Prepare print-out *)
prWin:= MG.CurWin;
MG.SetVisiScr;

(* Set position on the paper *)
MG.SetPos(x,y);

(* Print it *)
MG.CopyView(prWin);

(* New page *)
MG.FlushVisiScr;

Program Examples

GAHCopy1 makes a hardcopy of the monitor screen to the printer. It works with monochrome printers as well as color printers.

GAEps shows how graphics can be drawn on virtual screens and then printed. The program also shows how images of the type Image can be handled directly and be printed using MGImg.DrawImage. It also presents a large number of typefaces from the default font, as well as how the monitor screen can be "dumped" to the printer. The program can be compiled
Drivers D7-8  MGEps - Epson, IBM and NEC printers

with various printer modes and resolutions. GAEps works with both monochrome printers and color printers.

FuncPlot is a device independent example of function plotting. Output may be directed to screen, printers or plotters.

Family Data

deviceFamily  10000
deviceType    MG.rasterDevice
bitsPerPixel  1
maxColorPlane 2 for color printers
maxPaletteIndex 7 for color printers
maxPaletteColor 7 for color printers

Modes

IBM Pro          0  IBM Proprinter (9 pins)
epsMX           1  Epson MX80 II, Epson MX100 (9 pins)
epsFX           2  Epson FX, RX (9 pins)
epsJX           3  Epson JX (9 pins color)
epsLQ           4  Epson LQ (24 pins)
epsLQ25         5  Epson LQ25 (24 pins color)
ibmProX24      6  IBM Proprinter X24 (24 pins)
necP6           7  NEC P6 (24 pins)

maxDeviceMode  7

Printer modes are assigned using SetDeviceMode. The selection must be carried out before activating the device with SetDevice.

Error Codes

invalidLF  10001
invalidCol 10002
invalidRow 10003
invalidDeviceColor 10004

Specific error codes in \texttt{MGEps}.

## Colors

### Data Types

\begin{verbatim}
EpsPaletteColor = 0..maxPaletteColor;
\end{verbatim}

Palette colors for color printers (epsJX and epsLQ25).

\begin{verbatim}
EpsPaletteArray = ARRAY[0..maxPaletteIndex] OF EpsPaletteColor;
\end{verbatim}

\begin{verbatim}
EpsPaletteStruc = RECORD
    strucSize: Word;
    data     : EpsPaletteArray;
END;
\end{verbatim}

\begin{verbatim}
EpsPalette = ^EpsPaletteStruc;
\end{verbatim}

The \texttt{DevicePalette} structure for Epson and compatible color printers can be used when calling \texttt{MG.SetDevicePalette}.

\begin{verbatim}
EpsRGBArray = ARRAY EpsPaletteColor OF RGB;
\end{verbatim}

The RGB palette array for Epson and compatible color printers. Color printers use the following default palette:

\begin{verbatim}
RGB(  0,  0,  0),           black
RGB(  0,  0,666),           blue
RGB(  0,666,  0),           green
RGB(666,666,  0),           orange
RGB(  0,  0,  0),           red
RGB(666,  0,  0),           violet
RGB(999,999,333),           yellow
RGB(999,999,999))           white
\end{verbatim}
SetDeviceCall Functions

The Output Area - setSize

Constants

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size80x66</td>
<td>0</td>
<td>US format</td>
</tr>
<tr>
<td>size80x72</td>
<td>1</td>
<td>European format</td>
</tr>
<tr>
<td>size80xunlimited</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>size136x66</td>
<td>3</td>
<td>US format</td>
</tr>
<tr>
<td>size136x72</td>
<td>4</td>
<td>European format</td>
</tr>
<tr>
<td>size136xunlimited</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>maxSize</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Default value: size80x66 (US format).

This specifies the sizes of paper that are supported (in columns and lines). size80xunlimited means 80 characters in width and unlimited height (up to 16384 pixels). The number of columns depends on whether the printer can handle both landscape and portrait formats. The number of lines affects among other things page advancement. The constants are used when calling MG.SetDeviceCall. Here is how to set the European standard format:

MG.SetDeviceCall(pr, MG.setSize, MGEps.size80x72);

Data Types

PrintSizeSet = SET OF 0..maxSize;

Dot Density - setDensity

Constants

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dens60x72</td>
<td>0</td>
</tr>
<tr>
<td>dens120x72</td>
<td>1</td>
</tr>
<tr>
<td>dens120x72draft</td>
<td>2</td>
</tr>
<tr>
<td>dens60x60</td>
<td>3</td>
</tr>
</tbody>
</table>
The density is specified in pixels per inch both horizontally and vertically. N.B. that the densities permitted depend on which mode has been chosen. The constants are used when calling MG.SetDeviceCall. Here is how to set the density to 180 dots/inch:

MG.SetDeviceCall(pr,MG.setDensity,MGEps.dens180x180);

Data Types

DensitySet = SET OF 0..maxDens;

Typeface - setTypeFace

Constants

draft 0
courier 1
roman 2
sans 3
prestige 4

maxFace 4

Default value: draft.

This defines the default font's typeface. N.B. that each printer font (the default font using MGEps) can contain several different typefaces. All printers can handle draft while roman, sans and prestige, for example, can only be used with 24-pin printers. The constants are used when calling MG.SetDeviceCall. Here is how to assign the typeface to roman:

MG.SetDefaultFont;
MG.SetDeviceCall(pr,MG.setTypeFace,MGEps.roman);
Data Types

TypefaceSet = SET OF 0..maxFace;

Typeface Style - setTypeFaceStyle

Constants

<table>
<thead>
<tr>
<th>Typeface Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>upright</td>
<td>0</td>
</tr>
<tr>
<td>italic</td>
<td>1</td>
</tr>
<tr>
<td>outline</td>
<td>2</td>
</tr>
<tr>
<td>shadow</td>
<td>3</td>
</tr>
<tr>
<td>outlineShadow</td>
<td>4</td>
</tr>
<tr>
<td>maxStyle</td>
<td>4</td>
</tr>
</tbody>
</table>

Default value: **upright**

This defines the styles of the printer specific typefaces. Only certain printer modes support all styles. The constants are used when calling MG.SetDeviceCall. Here is how to set **roman italic**:

MG.SetDefaultFont;
MG.SetDeviceCall(pr,MG.setTypeFace,MGEps.roman);
MG.SetDeviceCall(pr,MG.setTypeFaceStyle,MGEps.italic);

Data Types

TypefaceStyleSet = SET OF 0..maxStyle;

Typeface Weight - setTypeFaceWeight

Constants

<table>
<thead>
<tr>
<th>Typeface Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>medium</td>
<td>0</td>
</tr>
<tr>
<td>bold</td>
<td>1</td>
</tr>
</tbody>
</table>

Default value: **medium**
The weight specifies the "boldness" of the printer typeface. Only certain printer modes can handle all the weights. The constants are used when calling `MG.SetDeviceCall`. Here is how to set `roman italic bold`:

```pascal
MG.SetDefaultFont;
MG.SetDeviceCall(pr,
    MG.setTypeFace,
    MGEps.roman);
MG.SetDeviceCall(pr,
    MG.setTypeFaceStyle,
    MGEps.italic);
MG.SetDeviceCall(pr,
    MG.setTypeFaceWeight,
    MGEps.bold);
```

### Data Types

`TypefaceWeightSet = SET OF 0..maxWeight;`

### Typeface Size - `setTypeFaceSize`

#### Constants

- `pica` 0
- `condPica` 1
- `expandPica` 2
- `condExpandPica` 3
- `elite` 4
- `condElite` 5
- `expandElite` 6
- `condExpandElite` 7
- `maxPitch` 7

Default value: `pica`.

This specifies the width of the printer typefaces. Only certain printer modes can handle all the widths. `pica` means 10 characters/inch, while `elite` means 12 characters/inch. The constants are used when calling `MG.SetDeviceCall`. Here is how to set `roman italic bold expandPica`:
Drivers D7-14

MGEps - Epson, IBM and NEC printers

MG.SetDefaultFont;
MG.SetDeviceCall(pr,
    MG.setTypeFace,
    MGEps.roman);
MG.SetDeviceCall(pr,
    MG.setTypeFaceStyle,
    MGEps.italic);
MG.SetDeviceCall(pr,
    MG.setTypeFaceWeight,
    MGEps.bold);
MG.SetDeviceCall(pr,
    MG.setTypeFaceSize,
    MGEps.expandPica);

Data Types

_TypefacePitchSet_ = SET OF 0..maxPitch;

The Character Set - setCharSet

Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibmMulti</td>
<td>0</td>
</tr>
<tr>
<td>maxCharSet</td>
<td>0</td>
</tr>
</tbody>
</table>

Default value: _ibmMulti_.

Only IBM's extended 8 bits character set can be used.

Data Types

_SetOfCharacterSet_ = SET OF 0..maxCharSet;
DeviceCall Functions

Mode Information - getDeviceModeInfo

Data Types

\[
\text{DeviceModeInfo} = \text{RECORD}
\]

\[
\begin{align*}
\text{sizing} & : \text{PrintSizeSet;} \\
\text{defaultSize} & : \text{Word;} \\
\text{differences} & : \text{DensitySet;} \\
\text{defaultDens} & : \text{Word;} \\
\text{maxColorPlane} & : \text{Word;} \\
\text{maxDeviceColor} & : \text{DeviceColor;} \\
\text{maxColorMask} & : \text{DeviceColor;} \\
\text{maxPaletteIndex} & : \text{Word;} \\
\text{maxPaletteColor} & : \text{EpsPaletteColor;} \\
\text{defaultPal} & : \text{EpsPaletteStruc;} \\
\text{rgbPal} & : \text{EpsRGBArray;} \\
\text{hasReverseLF} & : \text{BOOLEAN;} \\
\text{typefaces} & : \text{TypefaceSet;} \\
\text{defaultFace} & : \text{Word;} \\
\text{typefaceStyles} & : \text{TypefaceStyleSet;} \\
\text{defaultFaceStyle} & : \text{Word;} \\
\text{typefaceWeights} & : \text{TypefaceWeightSet;} \\
\text{defaultFaceWeight} & : \text{Word;} \\
\text{typefacePitches} & : \text{TypefacePitchSet;} \\
\text{defaultFacePitch} & : \text{Word;} \\
\text{charSets} & : \text{SetOfCharacterSet;} \\
\text{defaultCharSet} & : \text{Word;} \\
\end{align*}
\]

This summarizes the capabilities and the default settings for the printer (e.g., \texttt{ibmProX24}). The settings can be read by calling \texttt{MG.DeviceCall}, function \texttt{MG.getDeviceModeInfo}.

\[
\text{VAR}
\]

\[
\begin{align*}
\text{pr} & : \text{MG.Device;} \\
\text{prModeInfo} & : \text{MGEps.DeviceModeInfo;} \\
\end{align*}
\]

\[
\text{MG(DeviceCall(pr, MG.getDeviceModeInfo, SizeOf(prModeInfo), prModeInfo);}
\]
IF MGEps.roman IN prModeInfo.typefaces THEN BEGIN
    MG.SetDeviceCall(pr,
    MG.setTypeface,
    MGEps.roman);

..

Current Mode - getDeviceSettings

Data Types

```
DeviceSettings = RECORD
    size      : Word;
    dens      : Word;
    pins      : INTEGER;
    width     : INTEGER;
    height    : INTEGER;
    aspect    : Word;
    face      : Word;
    faceStyle : Word;
    faceWeight: Word;
    facePitch : Word;
    charSet   : Word;
END;
```

This describes the current settings of a particular printer mode. Current settings can be read by calling `MG.DeviceCall`, function `MG.getDeviceSettings`.

VAR
pr        : MG.Device;
prSettings: MGEps.DeviceSettings;

MG.DeviceCall(pr,
    MG.getDeviceSettings,
    SizeOf(prSettings),
    prSettings);

IF prSettings.dens = MGEps.dens180x180 THEN ..
## Summary of Printer Modes

X = is supported
(x) = is supported if the printer permits it.
X = normal setting

<table>
<thead>
<tr>
<th>deviceMode</th>
<th>ibmPro</th>
<th>epsMX</th>
<th>epsFX</th>
<th>epsJX</th>
<th>epsLQ</th>
<th>epsLQ25</th>
<th>ibmProX24</th>
<th>necP6</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviceId</td>
<td>10000</td>
<td>10001</td>
<td>10002</td>
<td>10003</td>
<td>10004</td>
<td>10005</td>
<td>10006</td>
<td>10007</td>
</tr>
<tr>
<td>maxColorPlane</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>maxDeviceColor</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>maxPaletteIndex</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>maxPaletteColor</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hasReverseLF</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>

### Orientation:

- **portrait**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **landscape**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X

### PrintSize:

- **size80x66**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **size80x72**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **size80xunlimited**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **size136x66**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **size136x72**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **size136xunlimited**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X

### Density:

- **dens60x72**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **dens120x72**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **dens120x72draft**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **dens60x60**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **dens120x60**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **dens180x180**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X

### TypeFace:

- **draft**
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **courier**
  - ibmPro: -
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **roman**
  - ibmPro: -
  - epsMX: -
  - epsFX: -
  - epsJX: -
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- **sans**
  - ibmPro: -
  - epsMX: -
  - epsFX: -
  - epsJX: -
  - epsLQ: -
  - epsLQ25: -
  - ibmProX24: -
  - necP6: -
- **prestige**
  - ibmPro: -
  - epsMX: -
  - epsFX: -
  - epsJX: -
  - epsLQ: -
  - epsLQ25: -
  - ibmProX24: -
  - necP6: -
<table>
<thead>
<tr>
<th>deviceMode</th>
<th>ibmPro</th>
<th>epsMX</th>
<th>epsFX</th>
<th>epsJX</th>
<th>epsLQ</th>
<th>epsLQ25</th>
<th>ibmProX24</th>
<th>necP6</th>
</tr>
</thead>
</table>

**TypeFaceStyle:**
- upright
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- italic
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- outline
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- shadow
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- outlineShadow
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X

**TypeFaceWeight:**
- medium
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- bold
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X

**TypeFacePitch:**
- pica
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- condPica
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- expandPica
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- condExpandPica
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- elite
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- condElite
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- expandElite
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X
- condExpandElite
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X

**CharacterSet:**
- ibmMulti
  - ibmPro: X
  - epsMX: X
  - epsFX: X
  - epsJX: X
  - epsLQ: X
  - epsLQ25: X
  - ibmProX24: X
  - necP6: X

**MGHPL** supports several different paper formats, resolutions from 75 dpi (dots/inch) up to 600 dpi (4/4M), and in-built or downloadable fonts, etc. The default installation has been chosen to work with older printer models and printers with no extra fonts. N.B. printout routines and printer modes must be chosen *before* activating the printer by **MG.SetDevice**:

```pascal
MG.LoadDevice(pr,'HPL.MGA'); (*real mode*)
MG.SetDeviceOutput (pr,MGOut.LPT1);
MG.SetDeviceMode (pr,MGHPL...);
MG.SetDeviceCall (pr,MG..,MGHPL..);
```

In the example above, the printout is done via **LPT1** (the parallel port). Remember that the laser printer is a "black-on-white" device!
Screens

The Visible Screen

Driver: HPL.MGV or HPL.MGA/HPL.XGA

The visible screen comprises the paper in the printer. The orientation can be set to portrait mode (default) or landscape mode. If the orientation is portrait, the paper's lower left corner has the position $x = 0, y = 0$. If the orientation is landscape, the paper's upper left corner has the position $x = 0, y = 0$. The sole drawing operation permitted is copying from the virtual and mono virtual screen respectively to the visible screen.

This means that all graphics must be drawn on a virtual or mono virtual screen and then be printed by copying to the visible screen using MG.CopyView or MG.CopyRect. Since images (MGImg.Image) are implemented using virtual screens, graphics can also be written out directly from images by calling MGImg.DrawImage or MGImg.DrawImagePart.

The only drawing mode permitted is replace. Positioning can be done freely on the drawing area, which means that screens or images can be copied to any position on the paper before the page is output. Of course you can define a drawing area which is of the same size as a full page, provided there is enough memory, and transfer this to the printer in a single copying operation.

Before activating the printer with MG.SetDevice, an output procedure must be installed. This is done by calling MG.SetDeviceOutput. If you want to change the default portrait orientation you must also do this before calling MG.SetDevice.

MG.FlushVisiScr is used to advance to a new page.

Mono virtual screens, and in some situations also virtual screens, can be in different graphics devices, provided that these have the same screen structure (1 bit/pixel) as HPL. In such cases, you can load HPL.MGV and save some memory.
Virtual Screens

Driver: HPL.MGA/HPL.XGA

All drawing modes and drawing operations can be used (1 bit/pixel). Copying to the visible screen is allowed, but not copying from the visible screen. Copying between virtual screens and mono virtual screens can also be done.

Moreover, printouts using hardware supported (as well as font cassettes) or downloaded printer fonts are allowed, but only in portrait mode. The internal character buffer contains a maximum of approximately 2500 characters of text which can be printed out with this font type.

Memory requirements: A full A4 page, printed with the maximum resolution (600 dpi using 4/4M models), requires approximately 3.7 MB memory! A full A4 page, printed with 300 dpi, requires approximately 1 MB memory. Instead of printing a full page at one go, you can use a smaller virtual screen and copy it several times to different positions on the same page. Furthermore, virtual screens can be allocated in virtual memory by including AutoVM in the USES list.

Mono Virtual Screens

The same data as for virtual screens.

Special Screens

Not available.

Line Style

Line styles in virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type MGPat.LineStyle16Struc. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

\[
\text{HPLLineStyle} = \wedge\text{MGPat.LineStyle16Struc};
\]

Generic line styles in MGPat can be used.
Pattern

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

```
PatternStruc = RECORD
    strucSize: Word; (* = 6 + patSize *)
    byteWidth: Word;
    height   : Word;
    pat      : ARRAY[1..patSize] OF Byte;
END;
```

```
HPLPattern = ^PatternStruc;
```

The size of the field `pat` is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often 8x8 bit patterns of the internal type `MGPat.Pattern8Struc` are used at the lower print densities and 16x16 bit patterns of the internal type `MGPat.Pattern16Struc` at the higher resolutions. See MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in `MGPat` can be used.

Printer Fonts

`MGHPL` can be used with bitmapped, stroke, outline and Bézier fonts. Moreover `MGHPL` can use certain printer resident (even cassette) or downloadable fonts (of the type `MG.deviceSymbols`) when printing in portrait mode.

The following fonts are supported:

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Style</th>
<th>Resident</th>
<th>Font file</th>
</tr>
</thead>
<tbody>
<tr>
<td>courier10</td>
<td>constant</td>
<td>normal</td>
<td>yes</td>
<td>default font</td>
</tr>
<tr>
<td>courier12</td>
<td>constant</td>
<td>normal</td>
<td>yes</td>
<td>HPLCOU12.DCH</td>
</tr>
<tr>
<td>linePrinter</td>
<td>constant</td>
<td>normal</td>
<td>yes</td>
<td>HPLL.P.DCH</td>
</tr>
<tr>
<td>helvetica</td>
<td>prop</td>
<td>bold</td>
<td>-</td>
<td>HPLHELV.DCH</td>
</tr>
</tbody>
</table>
Default font (portrait mode): **courier10** (also set by calling **MG.SetDefaultFont**).

Fonts that are not resident in the printer require a F-cassette or must first be downloaded to the printer.

The device specific fonts can only be used in portrait mode. All device specific fonts except the default font must be installed with **MG.LoadFont** (this loads the font header and width tables) and the desired .DCH-file. The printer must also support the installed font, otherwise the printout will be incorrect.

The number of characters that can be printed with device specific fonts is limited to approx. 2500 per screen.

### Printout Outline

The outline below shows how to draw with full resolution on the laser printer. In "MG - Copy Graphics", you will also find an example of how to make a hard copy of the monitor screen.

```pascal
USES MG, MGHPL, AutoVM;

VAR
    pr  : MG.Device;
    prScr: MG.Screen;
    prWin: MG.Window;
    ..
    MG.LoadDevice(pr,'HPL.MGA'); (*real mode*)
    MG.SetDeviceOutput (pr,MGOut.LPT1);
    (* Choose the printer orientation and configuration *)
    MG.SetOrientation (pr,..);
    MG.SetDeviceCall (pr,MG...,MGHPL...);
    ..
    MG.SetDevice (pr);
    MG.CreateMonoVirtScr (prScr,
                        MG.CurScrWidth,
                        MG.CurScrHeight);
```
MG.SetScr(prScr);

(* The paper is white, so we need to use black-on-white mode *)
MG.SetBlackOnWhite(TRUE);
MG.MatchForeground(MG.black);
MG.MatchBackground(MG.white);
MG.ClrWin; (* Fill with white *)

(* Draw graphics *)
.. prWin:= MG.CurWin;
MG.SetVisiScr;

(* Set the position on the paper for the lower left corner of the graphics *)
MG.SetPos(x,y);
MG.CopyView(prWin);
.. (* New page *)
MG.FlushVisiScr;

Program Examples

GAHCopy2 makes a hardcopy of the monitor screen to the printer. It works with all printer modes.

GAHPL shows how graphics can be drawn on virtual screens and then be printed out on the printer. The program also shows how images of the type MGImg.Image can be handled directly and printed using MGImg.DrawImage. It also presents a large number of HP device fonts (which require access to TimesRoman and Helvetica), as well as how the monitor screen can be "dumped" to the printer. The program can be compiled with various resolutions.

PCX Images

MGImg can read and write monochrome PCX images.
Family Data

deviceFamily 11001
deviceType rasterDevice
bitsPerPixel 1
maxDeviceColor 1
maxColorMask 1
maxColorPlane 0
maxPaletteIndex 1
maxPaletteColor 1
aspect 1000

courier10 0
courier12 1
linePrinter 2
helvetica 3
tmsRoman15 4
tmsRoman12 5
iTmsRoman12 6
bTmsRoman12 7

Modes

laserPlus 0
maxDeviceMode 0

MGHPL has only one mode, mode 0.

Error Codes

invalidCol 10002
invalidRow 10003
Colors

MGHPL supports only monochrome printers.

Data Types

HPLPaletteColor = 0..maxPaletteColor;

HPLPaletteArray =
ARRAY[0..maxPaletteIndex] OF HPLPaletteColor;

HPLPaletteStruc = RECORD
  strucSize: Word;
  data     : HPLPaletteArray;
END;

HPLPalette = ^HPLPaletteStruc;

The DevicePalette structure for HP LaserJet and compatible printers can be used when calling MG.SetDevicePalette.

SetDeviceCall Functions

The Output Area - setSize

Constants

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>legal</td>
<td>0</td>
<td>8.5&quot; x 14&quot;</td>
</tr>
<tr>
<td>letter</td>
<td>1</td>
<td>8.5&quot; x 11&quot;</td>
</tr>
<tr>
<td>A4</td>
<td>2</td>
<td>210 x 297 mm</td>
</tr>
<tr>
<td>B5</td>
<td>3</td>
<td>182 x 257 mm</td>
</tr>
<tr>
<td>maxSize</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Default value: A4 (European/international format).
This specifies the sizes of paper that are supported (in columns and lines). The constants are used when calling `MG.SetDeviceCall`. Here is how you set the US letter format:

```pascal
MG.SetDeviceCall(pr, MG.setSize, MGHPL.letter);
```

The paper orientation (portrait or landscape) is set by calling `MG.SetOrientation`.

### Data Types

```pascal
PrintSizeSet = SET OF 0..maxSize;
```

### Dot Density - setDensity

#### Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dens75</td>
<td>0</td>
<td>75 dots/inch</td>
</tr>
<tr>
<td>dens100</td>
<td>1</td>
<td>100 dots/inch</td>
</tr>
<tr>
<td>dens150</td>
<td>2</td>
<td>150 dots/inch</td>
</tr>
<tr>
<td>dens300</td>
<td>3</td>
<td>300 dots/inch</td>
</tr>
<tr>
<td>dens600</td>
<td>4</td>
<td>600 dots/inch</td>
</tr>
<tr>
<td>maxDens</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Default value: `dens75`.

The density is specified in dots per inch or dpi (the same resolution horizontally and vertically). The constants are used when calling `MG.SetDeviceCall`. The 600 dpi resolution is only allowed with HP LaserJet 4/4M and compatible printers.

The greater the resolution, the more memory needed and the longer the printout will take. Since the printing time increases quadratically to the resolution, it is often worthwhile to do test printouts using lower resolutions and to spare the full resolution until the final version is to be printed.

Here is how you set the resolution to 300 dots/inch:

```pascal
MG.SetDeviceCall(pr, MG.setDensity, MGHPL.dens300);
```
Data Types

DensitySet = SET OF 0..maxDens;

The Character Set - setCharSet

Constants

ibmMulti 0
roman8 1
maxCharSet 1

Default value: ibmMulti.

roman8 is Hewlett Packards own character set which is used internally by at least some of the laser printer models.

Data Types

SetOfCharacterSet = SET OF 0..maxCharSet;

DeviceCall Functions

Mode Information - getDeviceModeInfo

Data Types

DeviceModeInfo = RECORD
  sizes : PrintSizeSet;
  defaultSize : Word;
  densities : DensitySet;
  defaultDens : Word;
  charSets : SetOfCharacterSet;
  defaultCharSet : Word;
This summarizes the capabilities and the default settings for HP LaserJet (at present, there is only one mode). The settings can be read by calling MG.DeviceCall, function MG.getDeviceModeInfo.

```pascal
VAR
  pr        : MG.Device;
  prModeInfo: MGHPL.DeviceModeInfo;

MG.DeviceCall(pr,
               MG.getDeviceModeInfo,
               SizeOf(prModeInfo),
               prModeInfo);
```

Current Mode - getDeviceSettings

Data Types

```pascal
DeviceSettings = RECORD
  size   : Word;
  dens   : Word;
  width  : INTEGER;
  height : INTEGER;
  charSet: Word;
END;
```

This describes the current settings of a particular printer mode. Current settings can be read by calling MG.DeviceCall, function MG.getDeviceSettings.

```pascal
VAR
  pr        : MG.Device;
  prSettings: MGHPL.DeviceSettings;

MG.DeviceCall(pr,
               MG.getDeviceSettings,
               SizeOf(prSettings),
               prSettings);

IF prSettings.dens = MGHPL.dens600 THEN ..
The graphics driver MGHPGL (HPGL.MGV/HPGL.XGV) works with HPGL compatible pen plotters and other devices (laser printers, etc.) that emulate HPGL (= Hewlett-Packard Graphic Language). The driver has direct support for HP7475A (default mode) and HP7470 from Hewlett-Packard.

MGHPGL can use several different paper formats, pens, drawing speeds, fonts, landscape and portrait orientations etc. N.B. that printing routines and printer modes must be chosen before activating the plotter with MG.SetDevice:

```pascal
MG.LoadDevice(plotter,'HPGL.MGV'); (*real mode*)
MGOut.InitCOM2(MGOut.hardware,9600,8,1,MGOut.noParity);
MG.SetDeviceOutput(plotter,MGOut.COM2);
MG.SetDeviceMode(plotter,MGHPGL...);
MG.SetDeviceCall(plotter,MG..,MGHPGL..);
..MG.SetDevice(plotter);
MG.SetBlackOnWhite(TRUE);
..```

In the example above, the printout is done via COM2 (serial port 2) which is initialized to 9600 baud (initialization can also be carried out using the MODE command in DOS). Remember that the plotter is a "black-on-white" device!
Screens

The Visible Screen

Driver: HPGL.MGV/HPGL.XGV

The visible screen is the paper in the plotter. The paper's lower left corner has the position x = 0 and y = 0. See "Paper sizes" below for more data.

MGHPGL is a vector based device that does not support virtual, mono virtual or special screens. This means that all graphics must be drawn directly on the visible screen!

A printout procedure must be installed with MG.SetDeviceOutput before activating the plotter using MG.SetDevice. This is done with MG.SetDeviceOutput.

MG.FlushVisiScr is used to advance to a new page.

Limitations of MGHPGL

The only drawing modes permitted are replace and stamp.

The following graphics commands are not supported by MGHPGL:

- MG.PointDeviceColor, MGIW.PointDeviceColor,
  MGW.PointDeviceColor,
- MG.CurPointDeviceColor, MGIW.CurPointDeviceColor,
  MGW.CurPointDeviceColor,
- MG.CopyRect, MG.CopyView, MGIW.CopyRect, MGW.CopyRect,
- MG.FillArea, MG.FillPolygon, MG.FillEllipseSector,
  MGIW.FillPolygon, MGIW.FillEllipseSector, MGW.FillPolygon,
  MGW.FillEllipseSector,
- image handling in MGImg,
- input procedures in MGIO,
- mouse handling in MGMouse.

For fonts, the following applies: only stroke fonts (MG.strokeSymbols), unfilled outline fonts (MG.outlineSymbols) and unfilled Bézier fonts (MG.bezierSymbols) can be used. The same is true for symbols.
Virtual Screens
Not available.

Mono Virtual Screens
Not available.

Special Screens
Not available.

Line Style
Line styles are represented internally as

HPGLLineStyle = ^MGPat.HardwareLineStyleStruc;

HardwareLineStyleStruc = RECORD
  strucSize: Word;
  data : MGPat.GenericLineStyle;
END;

This means that only generic line styles (solid, dotted, dashed and dashedDotted) that are defined in MGPat can be used. User defined line styles can not be created.

Patterns
Patterns are represented internally as

HPGLPattern = ^MGPat.HardwarePatternStruc;
HardwarePatternStruc = RECORD
  strucSize: Word;
  data : MGPat.GenericPattern;
END;

This means that only generic patterns that are defined in MGPat can be used. User defined patterns can not be created.
Fonts

MGHPGL has no default font, but can use all stroke, outline and Bézier fonts (not bitmapped) that are loaded using MG.LoadFont.

Printout Stylesheets & Forms

VAR
    plotter: MG.Device;
    ..
MG.LoadDevice(plotter,'HPGL.MGV'); (*real mode*)
MG.SetDeviceOutput(plotter,MGout.COM1);

(* Choose printer modes etc *)
MG.SetDeviceMode(plotter, MGHPGL.hp7470);
MG.SetDeviceCall(plotter, MG..., MGHPGL...);
MG.SetOrientation(plotter, MG.portrait);
    ..
MG.SetDevice(plotter);

MG.SetBlackOnWhite(TRUE);
(* Draw graphics *)
MG.MatchForeground(MG.red);
MG.DrawLineTo(x,y);
    ..

Program Examples

GAHPGL draws a pie chart and prints text using HPGL plotters.

Family Data

deviceFamily  20000
deviceType    MG.vectorDevice
maxColorPlane  2
maxPaletteIndex  8
maxPaletteColor  8
bitsPerPixel  1
aspect        1000
Data Types

Density = INTEGER; (* plotter units/inch *)
Default value: 1021 plotter units/inch.

Error Codes

invalidX 10002
invalidY 10003
invalidPen 10004

Modes

hp7475A 0
hp7470 1
maxDeviceMode 1
Default: hp7475A.

Colors

Data Types

HPGLPaletteColor = Word; (*0..maxPaletteColor*);

HPGLPaletteArray = ARRAY[0..maxPaletteIndex] OF HPGLPaletteColor;
HPGLPaletteStruc = RECORD
    strucSize: Word;
    data : HPGLPaletteArray;
END;

HPGLPalette = ^HPGLPaletteStruc;

HPGLRGBArray = ARRAY HPGLPaletteColor OF RGB;

MGHPGL allows you to use up to eight plotter pens. The HP7475A plotter (mode \texttt{hp7475A}) can use a maximum of six pens. The HP7470 (mode \texttt{hp7470}) can take a maximum of eight pens. The default palette looks like this:

<table>
<thead>
<tr>
<th>Pen</th>
<th>\texttt{hp7475A}</th>
<th>\texttt{hp7470}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>- (&quot;white&quot;)</td>
</tr>
<tr>
<td>1</td>
<td>black</td>
<td>black</td>
</tr>
<tr>
<td>2</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>green</td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>orange</td>
<td>orange</td>
</tr>
<tr>
<td>5</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>6</td>
<td>violet</td>
<td>violet</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>cyan</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>yellow</td>
</tr>
</tbody>
</table>

There is no "Pen 0" in the pen tray.
SetDeviceCall Functions

Paper Sizes- MG.setSize

Constants

<table>
<thead>
<tr>
<th>Paper Size</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4</td>
<td>0</td>
</tr>
<tr>
<td>letter</td>
<td>1</td>
</tr>
<tr>
<td>A3</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>maxSize</td>
<td>3</td>
</tr>
</tbody>
</table>

Default value: A4.

This specifies the sizes of paper that are supported by MGHPGL (you have to check that a particular plotter is able to handle all sizes). The constants are used when calling MG.SetDeviceCall. Here is how to set the A3 format:

MG.SetDeviceCall(driver, MG.setSize, MGHPGL.A3);

The drawing area for the different paper formats are

<table>
<thead>
<tr>
<th>Paper Format</th>
<th>Drawing Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4</td>
<td>10000x7200</td>
</tr>
<tr>
<td>letter</td>
<td>10000x7200</td>
</tr>
<tr>
<td>A3</td>
<td>15200x10000</td>
</tr>
<tr>
<td>B</td>
<td>15200x10000</td>
</tr>
</tbody>
</table>

The printout can be oriented for landscape (default) or portrait.

The HP7470 supports only A4 and letter formats.

Data Types

PrintSizeSet = SET OF 0..maxSize;
Drawing Speeds - setSpeed

Constant

```
setSpeed 20
```

Data Types

```
Velocity = Word; (*0..127*) (* cm/s*)
```

Default: 38 cm/s.

This function is used to set the drawing speed by calling `MG.SetDeviceCall`. This is how you set the speed to 19 cm/s:

```
MG.SetDeviceCall(driver,MGHPGL.setSpeed,19);
```

Chord Angle - setSmoothness

Constant

```
setSmoothness 21
```

Data Types

```
ChordAngle = INTEGER; (* 0..179*) (* grader *)
```

Default: 1 degree.

This function is used for setting the chord angle when drawing circles and arcs. HPGL plotters approximate circles and arcs by using straight lines. The smaller the chord angle - the shorter the lines are and the better the "roundness" becomes. The chord angle is assigned by using `MG.SetDeviceCall`. Here is how to set a chord angle of 5 degrees:

```
MG.SetDeviceCall(driver,MGHPGL.setSmoothness,5);
```
DeviceCall Functions

Mode Information - MG.getDeviceModeInfo

Data Types

DeviceModeInfo = RECORD
  sizes            : PrintSizeSet;
  defaultSize      : Word;
  defaultDens      : Density;
  defaultVelocity  : Velocity;
  defaultChordAngle: ChordAngle;
  maxColorPlane    : Word;
  maxDeviceColor   : DeviceColor;
  maxColorMask     : DeviceColor;
  maxPaletteIndex  : Word;
  maxPaletteColor  : Word;
  defaultPal       : HPGLPaletteStruc;
  defaultRGBPal    : HPGLRGBArray;
END;

This summarizes the capabilities and default settings for a particular printer mode (e.g., hp7475A). The settings can be read by calling MG.DeviceCall, function MG.getDeviceModeInfo.

VAR
  plotter    : MG.Device;
  devModeInfo: MGHPGL.DeviceModeInfo;

MG.DeviceCall(plotter,
  MG.getDeviceModeInfo,
  SizeOf(devModeInfo),
  devModeInfo);
IF MGHPGL.A3 IN devModeInfo.sizes THEN BEGIN
  MG.SetDeviceCall(plotter,
    MG.setSize,
    MGHPGL.A3);
  ..
Current Mode - **MG.getDeviceSettings**

**Data Types**

```pascal
DeviceSettings = RECORD
  size      : Word;
  dens      : Density;
  speed     : Velocity;
  smoothness: ChordAngle;
  width     : INTEGER;
  height    : INTEGER;
  rgbPal    : HPGLRGBArray;
END;
```

This describes the current settings of a particular printer mode. Current settings can be read by calling **MG.DeviceCall**, function **MG.getDeviceSettings**.

```pascal
VAR
  plotter    : MG.Device;
  devSettings: MGHPGL.DeviceSettings;

MG.DeviceCall(plotter,
  MG.getDeviceSettings,
  SizeOf(devSettings),
  devSettings);
IF devSettings.size = MGHPGL.A4 THEN ..
```
The graphics driver MGTsh (HER.MGV, HER.MGA) works with Toshiba's plasma-screen portables (640x400 pixels) and compatible graphics adapters. MGTsh has only one mode with a resolution of 640x400 pixels and the colors black and "light" (orange). This is how you activate the Toshiba mode:

MG.LoadDevice(driver,'TSH.MGA');
MG.SetDevice(driver);

Screens

The Visible Screen
Driver: TSH.MGV or TSH.MGA

The visible screen comprises the monitor's screen. All drawing modes and drawing operations can be used (1 bit/pixel and 1 color plane). The width is 640 pixels and the height 400 pixels. The monitor screen's lower left corner has x = 0, y = 0. All drawing operations and drawing modes are allowed.

Virtual Screens
Driver: TSH.MGA

All drawing modes and drawing operations can be used (1 bit/pixel and 1 color plane). Copying to and from visible screens is allowed. Copying
between virtual screens and mono virtual screens can also be done.

Memory requirements: A virtual screen having the same dimensions as the visible screen, i.e. 640x400 pixels, requires 32 KB.

Mono Virtual Screens
The same data as for virtual screens.

Special Screens
Not supported.

Optimizing Speed
When copying between screens or within screens, the maximum performance is obtained if you remember the following points:

- avoid copying between x positions that force the bit pattern to be shifted. The distance in the x value should be an multiple of 8 pixels.

- use the drawing mode replace.

The same advice also applies when drawing images. Also try to output text characters at byte boundaries in the x direction, i.e. the x value ought to be a multiple of 8 (fast positions are obtained by using MG.GetFastXY).

Line Style
Line styles on virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type MGPat.LineStyle16Struc. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

TshLineStyle = ^MGPat.LineStyle16Struc;

Generic line styles in MGPat can be used.
Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

\[
\text{PatternStruc} = \text{RECORD}
\begin{align*}
\text{strucSize} : \text{Word}; \\
\text{byteWidth} : \text{Word}; \\
\text{height} : \text{Word}; \\
\text{pat} : \text{ARRAY}[1..\text{patSize}] \text{ OF Byte};
\end{align*}
\text{END;}
\]

\[
\text{TshPattern} = ^\text{PatternStruc};
\]

The size of the field \text{pat} is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often 8x8 bit patterns of the internal type \text{MGPat.Pattern8Struc} are used. See "MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in \text{MGPat} can be used.

Standard Symbols \text{MGTsh} define all \text{MultiGraphics} standard symbols - see "MG - Symbols". The symbols size varies, however the maximum size is 16x16 pixels.

Default Font

The default font has a constant width and height of 8x16 pixels. This gives 80 characters per line and 25 lines.

PCX Images

\text{MGIImg} can read and write monochrome PCX images.
Family Data

<table>
<thead>
<tr>
<th>deviceType</th>
<th>MG.rasterDevice</th>
</tr>
</thead>
<tbody>
<tr>
<td>visibleWidth</td>
<td>640</td>
</tr>
<tr>
<td>visibleHeight</td>
<td>400</td>
</tr>
<tr>
<td>bitsPerPixel</td>
<td>1</td>
</tr>
<tr>
<td>maxColorPlane</td>
<td>0</td>
</tr>
<tr>
<td>aspect</td>
<td>840</td>
</tr>
<tr>
<td>maxDeviceColor</td>
<td>1</td>
</tr>
<tr>
<td>maxColorMask</td>
<td>1</td>
</tr>
<tr>
<td>maxPaletteIndex</td>
<td>1</td>
</tr>
<tr>
<td>maxPaletteColor</td>
<td>1</td>
</tr>
</tbody>
</table>

Colors

TshPaletteColor = Word;

TshPaletteArray = ARRAY[0..maxPaletteIndex] OF TshPaletteColor;

TshPaletteStruc = RECORD
   strucSize: Word;
   data     : TshPaletteArray;
END;

TshPalette = ^TshPaletteStruc;

Toshiba graphics are monochrome, using the two device colors 0 and 1 and two palette colors 0 and 1.
The graphics driver **MGSVGA** (SVGA.MGV, SVGA.MGA/SVGA.XGA, SVGA.MGH/SVGA.XGH) works with "SuperVGA" display adapters in byte oriented 256 color modes (1 pixel = 1 byte, 1 color plane). **MGSVGA** supports all VESA compatible 256 color modes as well as SuperVGA adapters based on the TSENG4000 chip and the Trident 8900/8900C adapter.

**MGSVGA** has a basic mode with a resolution of 320x200, which is identical to the MCGA standard. This mode works with all SuperVGA adapters. **MGSVGA** also supports 640x480, 800x600, 1024x768, 1280x1024, etc. pixels depending on the hardware support. All modes require analog monitors. With 640x480, a VGA compatible monitor will work well, while 800x600, 1024x768, etc. require a multi-synch monitor (interlaced or non-interlaced).

This is how you activate the VESA mode 257 with 640x480 pixels:

```pascal
MG.LoadDevice(driver,'SVGA.MGA'); (*real mode*)
MG.SetDeviceMode(driver,MGSVGA.vesa640x480x256);
MG.SetDevice(driver);
```
Screens

The Visible Screen

Driver: SVGA.MGV, SVGA.MGA/SVGA.XGA or SVGA.MGH/SVGA.XGH.

The visible screen comprises the monitor's screen. All drawing modes and drawing operations can be used (8 bit/pixel and 1 color plane). The width of the basic mode is 320 pixels and the height 200 pixels. The monitor screen's lower left corner has $x = 0$ and $y = 0$.

**TSENG 4000 based 256 colors modes:**

Mode 1 (tseng640x480x256) has a resolution of 640x480 pixels.

Mode 2 (tseng800x600x256) has a resolution of 800x600 pixels.

Mode 3 (tseng1024x768x256) has a resolution of 1024x768 pixels.

**Trident 8900/8900C 256 colors modes:**

Mode 4 (trident640x480x256) has a resolution of 640x480 pixels.

Mode 5 (trident800x600x256) has a resolution of 800x600 pixels.

Mode 6 (trident1024x768x256) has a resolution of 1024x768 pixels.

**All 256 colors VESA modes, including the standard modes:**

Mode 256 (vesa640x400x256) has a resolution of 640x400 pixels (not supported by all VESA adapters).

Mode 257 (vesa640x480x256) has a resolution of 640x480 pixels.

Mode 259 (vesa800x600x256) has a resolution of 800x600 pixels.

Mode 261 (vesa1024x768x256) has a resolution of 1024x768 pixels.

Mode 263 (vesa1280x1024x256) has a resolution of 1280x1024 pixels (not supported by all VESA adapters).

**MGSVGA** also supports other 256 colors VESA modes that may exist on a particular system.
The only difference between SVGA.MGA/SVGA.MGH is the driver default font. See the "The Default Font".

You can attach a Hercules graphics card and a second monochrome monitor and display graphics on both screens in parallel. The dual monitor combination is perfect for debugging graphics applications.

Detection of SuperVGA adapters and modes

The procedures MG.DetectDevice, MG.DetectDeviceDriver and MG.AutoDevice will only report standard VGA mode even with a Super-VGA adapter. To get a complete view of all supported modes, you can call MG.GetNextDeviceModeInfo repeatedly. See the GAModes example.

Virtual Screens

Driver: SVGA.MGA/SVGA.XGA or SVGA.MGH/SVGA.XGH.

All drawing modes and drawing operations can be used (1 byte/pixel or 8 bits/pixel and 1 plane). Copying to and from the visible screen is allowed. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A virtual screen having the same dimensions as the visible screen in default mode, i.e. 320x200 pixels requires 64 KB. A virtual screen having the same dimensions as the visible screen in a 1024x768 mode requires not less than 786432 bytes (768 KB)! In real mode, virtual memory (preferable EMS or XMS) is mandatory (use VM or AutoVM).

Mono Virtual Screens

1 bit/pixel. Otherwise, the same data as for virtual screens.

Memory requirements: A mono virtual screen having the same dimensions as the visible screen in a 1024x768 mode requires 96 KB.

Special Screens

Not available.
Speed Optimization

The most important thing is to remember the simple mathematical truth that the increase of the data size of one-dimensional objects (lines, circles etc.) is proportional to their dimensions, while the increase in data size of two-dimensional objects (filled rectangles, circles, bitmaps, etc.) is proportional to the square of their dimensions! A block of 300x300 pixels requires 90000 bytes while 600x600 pixels requires 360000 bytes, that is four times more data! The time taken to draw the latter block is four times the time to draw the former block!

The drawing mode replace and a black background color is most efficient for most drawing operations.

Line Style

Line styles are represented internally as a 16-bits values (2 bytes) according to the data type MGPat.LineStyle16Struc. See "MGPat - Standard Patterns" for examples on how to create your own line styles.

```
SVGALineStyle = ^MGPat.LineStyle16Struc;
```

Generic line styles in MGPat can be used.

Patterns

Patterns are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

```
PatternStruc = RECORD
  strucSize: Word; (* = 6 + patSize *)
  byteWidth: Word;
  height   : Word;
  pat      : ARRAY[1..patSize] OF Byte;
END;
```

```
SVGAPattern = ^PatternStruc;
```

The size of the field pat is not fixed.
Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often 16x16 bits patterns of the internal type `MGPat.Pattern16Struct` are used. See MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in `MGPat` can be used.

**Standard Symbols**

`MGSVGA` defines all `MultiGraphics` standard symbols - see "MG - Symbols". The symbol's size varies, however, the maximum size is 24x24 pixels.

**The Default Font**

The default font differs depending on the chosen drivers.

The drivers SVGA.MGV/SVGA.MGA/SVGA.XGA use a default font with a fixed width and height of 8x16 pixels. This results in the following number of characters:

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Characters/line</th>
<th>Full lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>320x200</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>640x480</td>
<td>80</td>
<td>30</td>
</tr>
<tr>
<td>800x600</td>
<td>100</td>
<td>37</td>
</tr>
<tr>
<td>1024x768</td>
<td>128</td>
<td>48</td>
</tr>
<tr>
<td>1280x1024</td>
<td>160</td>
<td>64</td>
</tr>
</tbody>
</table>

This default font is appropriate for the lower resolutions.

The driver SVGA.MGH/SVGA.XGH uses a default font with a fixed width and height of 12x24 pixels. This results in the following number of characters:
## Drivers D11-6

### MGSVGA - Super VGA

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Characters/line</th>
<th>Full lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>320x200</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>640x480</td>
<td>53</td>
<td>20</td>
</tr>
<tr>
<td>800x600</td>
<td>66</td>
<td>25</td>
</tr>
<tr>
<td>1024x768</td>
<td>85</td>
<td>32</td>
</tr>
<tr>
<td>1280x1024</td>
<td>106</td>
<td>42</td>
</tr>
</tbody>
</table>

This default font is appropriate for the higher resolutions.

## PCX Images

**MGImg** can read and write both monochrome and 256-colors PCX images. PCX color images for SVGA have the same internal format as PCX images for MCGA modes and 8514/A but differ from images for VGA, EGA, Hercules, CGA etc. Also PCX images for EGA and VGA can be used, but they are converted to 256 color format when loaded.

### Family data

```plaintext
deviceFamily           $14 (* 20 *)
deviceType             MG.rasterDevice
bitsPerPixel           8
maxColorPlane          0
maxDeviceColor         255
maxColorMask           255
maxPaletteIndex        255
maxPaletteColor        63
```
## Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcga320x200x256</td>
<td>0</td>
</tr>
<tr>
<td>tseng640x480x256</td>
<td>1</td>
</tr>
<tr>
<td>tseng800x600x256</td>
<td>2</td>
</tr>
<tr>
<td>tseng1024x768x256</td>
<td>3</td>
</tr>
<tr>
<td>trident640x480x256</td>
<td>4</td>
</tr>
<tr>
<td>trident800x600x256</td>
<td>5</td>
</tr>
<tr>
<td>trident1024x768x256</td>
<td>6</td>
</tr>
<tr>
<td>vesa640x400x256</td>
<td>256</td>
</tr>
<tr>
<td>vesa640x480x256</td>
<td>257</td>
</tr>
<tr>
<td>vesa800x600x256</td>
<td>259</td>
</tr>
<tr>
<td>vesa1024x768x256</td>
<td>261</td>
</tr>
<tr>
<td>vesa1280x1024x256</td>
<td>263</td>
</tr>
<tr>
<td>supremeMode</td>
<td>255</td>
</tr>
</tbody>
</table>

The default mode is `mcga320x200x256`. Using `supremeMode`, you can automatically select the mode with the highest resolution available on a particular system:

```pascal
MG.LoadDevice(driver,'SVGA.MGA');
MG.SetDeviceMode(driver,MGSVGA.supremeMode);
MG.SetDevice(driver);
```

```pascal
visibleWidth : INTEGER;
visibleHeight: INTEGER;
aspect        : Word;
```
### Colors

**SVGAPaletteColor** = RECORD  (* 3 bytes *)
  r: Byte;
  g: Byte;
  b: Byte;
END;

**SVGAPaletteArray** =
  ARRAY[0..maxPaletteIndex] OF SVGAPaletteColor;

**SVGAPaletteStruc** = RECORD
  strucSize: Word;
  data     : SVGAPaletteArray;
END;

**SVGAPalette** = ^SVGAPaletteStruc;

SVGA can show 256 simultaneous colors. The device colors (color index) lies within the range of 0..255. The palette color for each device color may in turn be set as an RGB value with 64 levels of intensity (0..63) for red, green and blue respectively (data type **SVGAPaletteColor**). Each palette color may be chosen from among \(64^3 = 262144\) colors!

The default palette (**MG.SetDefaultPalette**) has been chosen so that the
first sixteen colors (0..15) corresponds to the predeclared RGB colors in the unit \texttt{MG}. These 16 colors have the same color indexes as the same colors for EGA and VGA. The remaining colors (16..255) are selected so that the green intensity changes using small increments, while red and blue change using larger increments. The secret behind this is to utilize the differences in the eye's sensitivity to red, green and blue colors. Both device colors 15 and 255 are set to white.

The following is an example of reading the current palette, how the color index 0 is set to white and color index 255 is set to black:

\begin{verbatim}
VAR
  SVGAPal: MGSVGA.SVGAPalette;
  ..
MG.GetDevicePalette(MG.DevicePalette(SVGAPal));
WITH SVGAPal^ DO BEGIN
  WITH data[0] DO BEGIN
    r:= 63; g:= 63; b:= 63;
  END;
  WITH data[255] DO BEGIN
    r:= 0; g:= 0; b:= 0;
  END;
END;
MG.SetDevicePalette(MG.DevicePalette(SVGAPal));
  ..
MG.DisposeDevicePalette(MG.DevicePalette(SVGAPal));
\end{verbatim}

Of course, you could equally well have used the hardware independent palette handling in \texttt{MultiGraphics}.

\section*{DeviceCall-functions}

\subsection*{Detection of adapter - MG.detectDeviceMode}

By calling \texttt{MG.DeviceCall}, function \texttt{MG.detectDeviceMode}, you will be able to detect the 256 color VESA mode with the highest resolution or hardware type of the installed SuperVGA adapter. You can call it like this:
VAR
  driver     : MG.Device;
  adapterMode: Word;
  ..
  MG.LoadDevice(driver,'SVGA.MGA'); (*reellt läge*)
  MG.DeviceCall(driver,
    MG.detectDeviceMode,
    SizeOf(adapterMode),adapterMode);
  MG.SetDeviceMode(driver,adapterMode);
  MG.SetDevice(driver);

The **DeviceCall** call always returns the mode with the **highest resolution** of
the detected VESA modes or SuperVGA adapter (**tseng640x480x256**, etc).

You will get the same result with **MG.SetDeviceMode(driver,MGSVGA.supremeMode)** - see explanation above.
The graphics driver **MGPJet** (PJET.MGV, PJET.MGA/PJET.XGA, PJET.MGH/PJET.XGH) works with Hewlett Packard's PaintJet ink jet printers. The driver has support for both black & white output as well as color printing.

**MGPJet** can use portrait and landscape orientations as well as several different paper formats, resolutions, fonts, etc. N.B. that printing routines and printer modes must be selected *before* activating the printer by using **MG.SetDevice**:

```pascal
MG.LoadDevice(pr,'PJET.MGA'); (*real mode*)
MG.SetDeviceOutput(pr,MGOut.LPT1);
MG.SetDeviceMode(pr,MGPJet...);
MG.SetOrientation(pr,MG...);
MG.SetDeviceCall(pr,MG..,MGPJet..);
..
MG.SetDevice(pr);
MG.SetBlackOnWhite(TRUE)..
```

In the example above, the printout is done via **LPT1** (the parallel port). Remember that the printer is a "black-on-white" device!
Screens

The Visible Screen

Driver: PJET.MGV, PJET.MGA/PJET.XGA or PJET.MGH/PJET.XGH

The visible screen comprises the paper in the printer. If the orientation is portrait, the paper's lower left corner has the position x = 0, y = 0. If the orientation is landscape the paper's upper left corner has the position x = 0, y = 0. Default is portrait mode.

The only drawing operation permitted is copying from virtual and mono virtual screens respectively to the visible screen. This means that all graphics must be drawn on virtual or mono virtual screens and then afterwards be printed by copying to the visible screen with MG.CopyView or MG.CopyRect. Since images (Image) are implemented using virtual screens, the graphics can also be printed directly from images with MGImg.DrawImage or MGImg.DrawImagePart.

The only drawing mode permitted is replace. The PaintJet printer as well as the MGPJet driver support both forward and backward paper feed. However, it is recommended to restrict paper movements to decreasing y values (if portrait orientation) or increasing x values (if landscape orientation), since backward paper feed may smear out printings that have not dried. It is therefore advisable to make use of several consecutive copying operations so that you start by copying the virtual screen that is to be placed at the top of the paper, and so on. Of course, you can you create a large virtual screen that contains a full page and then copy the whole screen in one go, but this will increase the memory demands..

Before activating the printer by calling MG.SetDevice, a printing procedure must be installed. This is done by using MG.SetDeviceOutput.

MG.FlushVisiScr is used to advance to a new page.

Virtual and mono virtual screens can be in different graphics devices, provided that these have the same screen structure (bits/pixel, etc). In such cases, you can load PJET.MGV and save some memory.
Virtual Screens

Driver: PJET.MGA/PJET.XGA or PJET.MGH/PJET.XGH.

All drawing modes and drawing operations can be used (1 bit/pixel, 1 plane in monochrome modes, up to 4 planes when using color modes with resolution 90 dpi (dots/inch), and 3 planes with resolution 180 dpi). Copying to the visible screen is allowed, but not copying from visible screen. Copying to and from virtual screens and mono virtual screens can also be done.

Memory requirements: when using high resolution (180 dpi) and three color planes, approximately 1.1 MB memory is needed for a complete A4 page. Using 90 dpi and four color planes, approx. 400 KB is needed. It is recommended to work in protected mode. In real mode it is often necessary to place virtual screens in expanded EMS, extended XMS or on disk. You can do this by including AutoVM in the USES list.

Mono Virtual Screens

All drawing modes and drawing operations can be used. Copying to the visible screen is allowed, but not copying from visible screen. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: Approximately 380 KB when creating a full page using 180 dpi. Approximately 95 KB when creating a full page using 60 dpi.

Special Screens

Not available.

Colors

The default setup is two simultaneous colors (normally black and white). The number of colors may be changed by calling MG.SetDeviceMode to 4, 8 or 16 colors (see the constants pjet4Col, pjet8Col, pjet16Col). The multi-color modes are plane oriented with 2, 3 or 4 color planes. At the resolution 180 dpi, the printer can only print 8 fixed colors (see "Colors" below), while 90 dpi gives more freedom to choose the palette.
Line Styles

Line styles in virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type \texttt{MGPat.LineStyle16Struc}. See "MGPat - Standard Patterns" for examples of user defined line styles.

\texttt{PaintJetLineStyle} = \texttt{^MGPat.LineStyle16Struc;}

Bits set to one are drawn using the current foreground color while those set to zero are drawn with the background color.

Generic line styles in \texttt{MGPat} can be used.

Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

\texttt{PatternStruc} = \texttt{RECORD}
\begin{verbatim}
  strucSize: Word; (* = 6 + patSize *)
  byteWidth: Word;
  height   : Word;
  pat      : ARRAY[1..patSize] OF Byte;
END;
\end{verbatim}

\texttt{PaintJetPattern} = \texttt{^PatternStruc;}

The size of the field \texttt{pat} is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often we use 8x8 bit patterns of type \texttt{MGPat.Pattern8Struc} at 90 dpi and 16x16 patterns of type \texttt{MGPat.Pattern16Struc} at 180 dpi. See "MGPat - Standard Patterns" for declarations and examples of user defined patterns.

Generic patterns in \texttt{MGPat} can be used.
Standard Symbols

MGPJet defines the standard symbols 0..6 in accordance with the description in "MG - Symbols". The symbols do not have masks. The symbol size is specific for each symbol, but the maximum size is 24x24 pixels.

Printer Fonts

MGPJet can be used with bitmapped, stroke, outline fonts and Bézier fonts.

Default fonts:

PJET.MGV, PJET.MGA/PJET.XGA: 8x16 fixed, bitmapped font.

PJET.MGH/PJET.XGH: 12x24 fixed, bitmapped font.

PCX Images

The MGImg unit can both read and write monochrome and 2, 4, 8 and 16-colors PCX images. Color PCX images for PaintJet have the same internal format as color PCX images for EGA and VGA. However, if you run the printer in 180 dpi mode it uses only the information of the first three color planes.

Program Outlines

The outlines below show how to draw with full resolution on the printer. In "MG - Copy Graphics" you will also find an example of how to make a hard copy of the monitor screen.
Monochrome Graphics

USES MG,MGPJet,AutoVM;

VAR
    pr   : MG.Device;
    prScr: MG.Screen;
    prWin: MG.Window;

.. MG.LoadDevice(pr,'PJET.MGA'); (*real mode*)
MG.SetDeviceOutput(pr,MGOut.COM1);

(* Select printer modes etc *)
MG.SetDeviceMode(pr,MGPJet.pjet2Col);
MG.SetOrientation(pr,mg...);
MG.SetDeviceCall(pr,MGsetDensity,MGPJet.dens180);
MG.SetDeviceCall(pr,mg...,MGPJet...);
.. MG.SetDevice(pr);

MG.CreateMonoVirtScr(prScr,
    MG.CurScrWidth,
    MG.CurScrHeight);

MG.SetScr(prScr);

(* Paper is white so we need to use black-on-white mode *)
MG.SetBlackOnWhite(TRUE);
MG.MatchBackground(MG.white);
MG.MatchForeground(MG.black);
MG.ClrWin; (* Fill screen with white *)

(* Draw graphics *)
.. (* Prepare print-out *)
prWin:= MG.CurWin;
MG.SetVisiScr;

(* Set position on the paper for the lower left corner of the graphics *)
MG.SetPos(x,y);

(* Now print *)
MG.CopyView(prWin);

(* New page *)
MG.FlushVisiScr;
Color Graphics

USES MG,MGPJet,AutoVM;

VAR
    pr   : MG.Device;
    prScr: MG.Screen;
    prWin: MG.Window;
    ..
MG.LoadDevice(pr,'PJET.MGA'); (*real mode*)
MG.SetDeviceOutput (pr,MGOut.COM1);
 (* Select printer modes etc *)
MG.SetDeviceMode (pr,MGPJet.pjet16Col); (* 16 colors *)
MG.SetOrientation(pr,MG...);
MG.SetDeviceCall (pr,MGsetDensity,MGPJet.dens90);
MG.SetDeviceCall (pr,MG...,MGPJet...);
MG.SetDevice (pr);
    ..
MG.CreateVirtScr (prScr,
    MG.CurScrWidth,
    MG.CurScrHeight,
    MG.CurMaxColorMask);

MG.SetScr (prScr);

(* Paper is white so we need to use black-on-white mode *)
MG.SetBlackOnWhite(TRUE);
MG.MatchBackground (MG.white);
MG.ClrWin; (* Fill screen with white *)
MG.MatchForeground (MG.red);

(* Draw graphics *)
   ..
(* Prepare print-out *)
prWin:= MG.CurWin;
MG.SetVisiScr;

(* Set position on the paper *)
MG.SetPos(x,y);

(* Print it *)
MG.CopyView(prWin);

(* New page *)
MG.FlushVisiScr;
Program Examples

GAHCopy3 makes a hardcopy of the monitor screen to the printer. It works with monochrome as well as color modes.

GAPJet shows how graphics can be drawn on virtual screens and then printed. The program also shows how images of the type Image can be handled directly and be printed using MGImg.DrawImage. The program can be compiled with various printer modes and resolutions.

FuncPlot is a device independent example of function plotting. Output may be directed to screen, printers or plotters.

Family Data

deviceFamily 11210
deviceType MG.rasterDevice
bitsPerPixel 1
aspect 1000
maxColorPlane 3
maxPaletteIndex 1

Modes

pjet2Col 0 default
pjet4Col 1
pjet8Col 2
pjet16Col 3

maxDeviceMode 3

Printer modes are assigned using SetDeviceMode. The selection must be carried out before activating with SetDevice.
Error Codes

invalidCol 10002
invalidRow 10003

Specific error codes in MGPJet.

Colors

Constants

minRed 4
maxRed 90
minGreen 4
maxGreen 88
minBlue 6
maxBlue 85

Data Types

PJetPaletteColor = RECORD
  r: Byte; (* minRed .. maxRed *)
  g: Byte; (* minGreen..maxGreen *)
  b: Byte; (* minBlue..maxBlue *)
END;

Palette colors for PaintJet.

PJetPaletteArray = ARRAY[0..maxPaletteIndex] OF PJetPaletteColor;

PJetPaletteStruc = RECORD
  strucSize: CARDINAL;
  data     : PJetPaletteArray;
END;
PJetPalette = ^PJetPaletteStruc;

The DevicePalette structure for PaintJet and compatible color printers can be used when calling MG.SetDevicePalette.

As a default the following palettes are used:

<table>
<thead>
<tr>
<th>Device color</th>
<th>pjet2Col</th>
<th>pjet4Col</th>
<th>pjet8Col</th>
<th>90 dpi pjet16Col</th>
<th>180 dpi pjet16Col</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>black</td>
<td>black</td>
<td>black</td>
<td>black</td>
<td>black</td>
</tr>
<tr>
<td>1</td>
<td>white</td>
<td>red</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>green</td>
<td>green</td>
<td>green</td>
<td>green</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>white</td>
<td>cyan</td>
<td>cyan</td>
<td>cyan</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>red</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>magenta</td>
<td>magenta</td>
<td>magenta</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>yellow</td>
<td>brown</td>
<td>yellow</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>white</td>
<td>light gray</td>
<td>white</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>dark gray</td>
<td>black</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>light blue</td>
<td>blue</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>light green</td>
<td>green</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>light cyan</td>
<td>cyan</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>light red</td>
<td>red</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>light mag</td>
<td>magenta</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>yellow</td>
<td>yellow</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>white</td>
<td>white</td>
</tr>
</tbody>
</table>
SetDeviceCall Functions

The Output Area - setSize

Constants

- \texttt{size80x66} 0 US format
- \texttt{size80x72} 1 European format
- \texttt{size80xunlimited} 2
- \texttt{maxSize} 5

Default value: \texttt{size80x72} (European format).

This specifies the sizes of paper that are supported (in columns and lines). \texttt{size80xunlimited} means 80 characters in width and unlimited height (up to 16384 pixels). The number of columns depends on whether the printer can handle both landscape and portrait formats. The constants are used when calling \texttt{MG.SetDeviceCall}. Here is how to set the US standard format:

\begin{verbatim}
MG.SetDeviceCall(pr, MG.setSize, MGPJet.size80x66);
\end{verbatim}

Data Types

\texttt{PrintSizeSet = SET OF 0..maxSize;}

Dot Density - setDensity

Constants

- \texttt{dens90} 0
- \texttt{dens180} 1
- \texttt{maxDens} 1

Default value: \texttt{dens90}.

The density is specified in dots per inch (dpi). The constants are used when calling \texttt{MG.SetDeviceCall}. Here is how to set the density to 180 dpi:

\begin{verbatim}
MG.SetDeviceCall(pr, MG.setDensity, MGPJet.dens180);
\end{verbatim}
Drivers D12-12  MGPJet - HP PaintJet Color Printer

MG.SetDeviceCall(pr, MG.setDensity, MGPJet.dens180);

Data Types

\[
\text{DensitySet} = \text{SET OF } 0..\text{maxDens};
\]

Data Compression - MGPJet.setCompression

Constants

\[
\text{setCompression} = 20
\]

Used as a parameter to \text{MG.SetDeviceCall} to select transmission mode to the printer (compressed or uncompressed).

\[
\begin{align*}
\text{noCompression} & = 0 \\
\text{runLengthCompression} & = 1 \\
\text{maxCompression} & = 1
\end{align*}
\]

Default value: \text{noCompression}.

This function is used to select the transmission mode from the computer to the printer. Compressed run-length encoding gives faster output, especially if the graphics contain large mono colored areas. The default setup is with compression disabled.

Data Type

\[
\text{CompressionSet} = \text{SET OF } 0..\text{maxCompression};
\]
DeviceCall Functions

Mode Information - getDeviceModeInfo

Data Types

DeviceModeInfo = RECORD
  sizes              : PrintSizeSet;
  defaultSize        : Word;
  densities          : DensitySet;
  defaultDens        : Word;
  compressions       : CompressionSet;
  defaultCompression : Word;
  maxColorPlane      : Word;
  maxDeviceColor     : MG.DeviceColor;
  maxColorMask       : MG.DeviceColor;
  maxPaletteIndex    : Word;
END;

This summarizes the capabilities and the default settings for a specific device mode (for example pjet16Col). The settings can be read by calling MG.DeviceCall, function MG.getDeviceModeInfo.

VAR
  pr        : MG.Device;
  prModeInfo: MGPJet.DeviceModeInfo;

MG.DeviceCall(pr,
  MG.getDeviceModeInfo,
  SizeOf(prModeInfo),
  prModeInfo);

IF MGPJet.dens180 IN prModeInfo.densities THEN BEGIN
  MG.SetDeviceCall(pr,
    MG.setDensity,
    MGPJet.dens180);
..
Current Mode - getDeviceSettings

Data Types

\[
\begin{align*}
\text{DeviceSettings} & = \text{RECORD} \\
\text{size} & : \text{Word}; \\
\text{dens} & : \text{Word}; \\
\text{compression} & : \text{Word}; \\
\text{width} & : \text{INTEGER}; \\
\text{height} & : \text{INTEGER}; \\
\text{END};
\end{align*}
\]

This describes the current settings of a particular printer mode. Current settings can be read by calling \text{MG.DeviceCall}, function \text{MG.getDeviceSettings}.

\begin{verbatim}
VAR 
  pr        : MG.Device; 
  prSettings: MGPJet.DeviceSettings;

MG.DeviceCall(pr, 
  MG.getDeviceSettings, 
  SizeOf(prSettings), 
  prSettings); 
IF prSettings.dens = MGPJet.dens180 THEN ..
\end{verbatim}
The graphics driver **MGDJet** (DJET.MGV, DJET.MGA/DJET.XGA, DJET.MGH/DJET.XGH) works with Hewlett Packard's family of DeskJet printers, i.e. DeskJet, DeskJet PLUS, DeskJet 500 and DeskJet 500C/550C. The driver has support for both black & white output as well as color printing (500C/550C only).

**MGDJet** can use portrait and landscape orientations as well as several different paper formats, resolutions, fonts, etc. N.B. that printing routines and printer modes must be selected *before* activating the printer by using **MG.SetDevice**:

```pascal
MG.LoadDevice(pr,'DJET.MGA'); (*real mode*)
MG.SetDeviceOutput(pr,MGOut.LPT1);
MG.SetDeviceMode(pr,MGDJet...);
MG.SetOrientation(pr,MG..);
MG.SetDeviceCall(pr,MG..,MGDJet..);
..
MG.SetDevice(pr);
MG.SetBlackOnWhite(TRUE).. 
```

In the example above, the printout is done via **LPT1** (the parallel port). Remember that the printer is a "black-on-white" device!

You may use the "Quality" button on the printer to select fast "Draft" mode or "High" quality mode.
Screens

The Visible Screen

Driver: DJET.MGV, DJET.MGA/DJET.XGA or DJET.MGH/DJET.XGH.

The visible screen comprises the paper in the printer. If the orientation is portrait, the paper's lower left writeable corner has the position $x = 0, y = 0$. Due to paper feed restrictions, there is always an unwriteable bottom margin of approximately $1/2"$ (12 mm). If the orientation is landscape, the paper's upper left corner has the position $x = 0, y = 0$. Default is portrait mode.

The only drawing operation permitted is copying from virtual and mono virtual screens respectively to the visible screen. This means that all graphics must be drawn on virtual or mono virtual screens and then printed by copying to the visible screen with `MG.CopyView` or `MG.CopyRect`. Since images (Image) are implemented using virtual screens, the graphics can also be printed directly from images with `MGImg.DrawImage` or `MGImg.DrawImagePart`.

The only drawing mode permitted is replace. The DeskJet printer does not support backward paper feed, so you have to restrict paper movements to decreasing y values (if portrait orientation) or increasing x values (if landscape orientation). It is therefore advisable to make use of several consecutive copying operations so that you start by copying the virtual screen that is to be placed at the top of the sheet of paper, and so on. There is of course nothing to prevent you (except greater memory requirements) from creating a virtual or mono virtual screen that contains the complete sheet of paper and then printing the page in one go.

Before activating the printer by using `MG.SetDevice`, a printing routine must be installed. This is done by calling `MG.SetDeviceOutput`.

`MG.FlushVisiScr` is used to advance to a new page.

Virtual and mono virtual screens can be in different graphics devices, provided that these have the same screen structure (bits/pixel, etc). In such cases, you can load DJET.MGV and save some memory.
Virtual Screens

Driver: DJET.MGA/DJET.XGA or DJET.MGH/DJET.XGH.

All drawing modes and drawing operations can be used (1 bit/pixel, 1 plane in monochrome modes, up to 3 planes when using color modes (DeskJet 500C/550C only) with a maximum resolution of 300 dpi). Copying to the visible screen is allowed, but not copying from visible screen. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A full A4 page printed with the highest resolution (180 dpi) and three color planes, requires approximately 3 MB memory! Using 75 dpi and three color planes, approx. 185 KB is needed. It is recommended to work in protected mode. In real mode it is often necessary to place virtual screens in expanded EMS, extended XMS or on disk. You can do this by including AutoVM in the USES list.

Mono Virtual Screens

All drawing modes and drawing operations can be used. Copying to the visible screen is allowed, but not copying from visible screen. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: Approximately 1 MB when printing a full page using 300 dpi. Approximately 60 KB when printing a full page using 75 dpi.

Special Screens

Not available.

Colors

The default setup color is black (black ink) using djetBlack mode. The number of colors may be changed to 2, 4 or 8 colors by calling MG.SetDeviceMode (see the constants djet2Col, djet4Col, pdjet8Col). The multi-color modes are plane oriented with 1, 2 or 3 color planes. There are only 8 distinct colors in the repertoire of DeskJet 500C/550C but their position in the palette may be changed.
Line Styles

Line styles in virtual and mono virtual screens are represented internally as 16-bits values (2 bytes) according to the data type `MGPat.LineStyle16Struc`. See "MGPat - Standard Patterns" for examples of user defined line styles.

```
DeskJetLineStyle = ^MGPat.LineStyle16Struc;
```

Bits set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Generic line styles in `MGPat` can be used.

Patterns

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is a multiple of 8 (i.e. always full bytes). The height can be one or more lines.

```
PatternStruc = RECORD
  strucSize: Word; (* = 6 + patSize *)
  byteWidth: Word;
  height   : Word;
  pat      : ARRAY[1..patSize] OF Byte;
END;
```

```
DeskJetPattern = ^PatternStruc;
```

The size of the field `pat` is not fixed.

Bits that are set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often we use 8x8 bit patterns of type `MGPat.Pattern8Struc` at 75 - 100 dpi and 16x16 patterns of type `MGPat.Pattern16Struc` at 150 - 300 dpi. See "MGPat - Standard Patterns" for declarations and examples of user defined patterns.

Generic patterns in `MGPat` can be used.
Standard Symbols

MGDJet defines the standard symbols 0..6 in accordance with the description in "MG - Symbols". The symbols do not have masks. The symbol size is specific for each symbol, but the maximum size is 24x24 pixels.

Printer Fonts

MGDJet can be used with bitmapped, stroke, outline and Bézier fonts.

Default fonts:

DJET.MGV, DJET.MGA/DJET.XGA: 8x16 fixed, bitmapped font.

DJET.MGH/DJET.XGH: 12x24 fixed, bitmapped font.

PCX Images

The MGIImg unit can both read and write monochrome and 2, 4 and 8-color PCX images. Color PCX images for DeskJet 500C/550C have the same internal format as color PCX images for EGA and VGA, but a maximum of three color planes are used.

Program Outlines

The outlines below show how to draw with full resolution on the printer. In "MG - Copy Graphics" you will also find an example of how to make a hard copy of the monitor screen.
Drivers D13-6  MGDJet - HP DeskJet Printers

Monochrome Graphics using the Black Ink Cartridge

USES MG, MGDJet, AutoVM;

VAR
   pr : MG.Device;
   prScr: MG.Screen;
   prWin: MG.Window;
  ..
MG.LoadDevice(pr, 'DJet.MGA'); (*real mode*)
MG.SetDeviceOutput(pr, MGOut.LPT1);

(* Select printer modes etc *)
MG.SetDeviceMode(pr, MGDJet.djetBlack);
MG.SetOrientation(pr, MG...);
MG.SetDeviceCall(pr, MGsetDensity, MGDJet.dens150);
MG.SetDeviceCall(pr, MG..., MGDJet...);
  ..
MG.SetDevice(pr);

MG.CreateMonoVirtScr(prScr,
   MG.CurScrWidth,
   MG.CurScrHeight);

MG.SetScr(prScr);

(* Paper is white so we need to use black-on-white mode *)
MG.SetBlackOnWhite(TRUE);
MG.MatchBackground(MG.white);
MG.ClrWin; (* Fill screen with white *)
MG.MatchForeground(MG.black);

(* Draw graphics *)
  ..
(* Prepare print-out *)
prWin:= MG.CurWin;
MG.SetVisiScr;

(* Set position on the paper for the lower left corner of the graphics *)
MG.SetPos(x, y);

(* Now print *)
MG.CopyView(prWin);

(* New page *)
MG.FlushVisiScr;

MultiGraphics & Borland Pascal/Turbo Pascal  DATABITEN
**Color Graphics**

USES MG,MGDJet,AutoVM;

VAR
  pr   : MG.Device;
  prScr: MG.Screen;
  prWin: MG.Window;
  ..
MG.LoadDevice(pr,'DJet.MGA'); (*real mode*)
MG.SetDeviceOutput(pr,MGOut.LPT1);

(* Select printer modes etc. *)
MG.SetDeviceMode(pr,MGDJet.djet8Col); (* 8 colors *)
MG.SetOrientation(pr,MG...);
MG.SetDeviceCall(pr,MGsetDensity,MGDJet.dens300);
MG.SetDeviceCall(pr,MG... ,MGDJet...);
MG.SetDevice(driver);
  ..
MG.CreateVirtScr(prScr,
  MG.CurScrWidth,
  MG.CurScrHeight,
  MG.CurMaxColorMask);

MG.SetScr(prScr);

(* Paper is white so we need to use black-on-white mode *)
MG.SetBlackOnWhite(TRUE);
MG.MatchBackground(MG.white);
MG.ClrWin; (* Fill screen with white *)
MG.MatchForeground(MG.red);

(* Draw graphics *)
  ..
(* Prepare print-out *)
prWin:= MG.CurWin;
MG.SetVisiScr;

(* Set position on the paper *)
MG.SetPos(x,y);

(* Print it *)
MG.CopyView(prWin);

(* New page *)
MG.FlushVisiScr;
Program Examples

**GAHCopy4** makes a hardcopy of the monitor screen to the printer. It works with monochrome as well as color modes.

**GADJet** shows how graphics can be drawn on virtual screens and then printed. The program also shows how images of the type **Image** can be handled directly and be printed using **MGImg.DrawImage**. The program can be compiled with various printer modes and resolutions.

**FuncPlot** is a device independent example of function plotting. Output may be directed to screen, printers or plotters.

Family Data

<table>
<thead>
<tr>
<th>deviceFamily</th>
<th>11200</th>
</tr>
</thead>
<tbody>
<tr>
<td>deviceType</td>
<td>MG.rasterDevice</td>
</tr>
<tr>
<td>bitsPerPixel</td>
<td>1</td>
</tr>
<tr>
<td>aspect</td>
<td>1000</td>
</tr>
<tr>
<td>maxColorPlane</td>
<td>2</td>
</tr>
<tr>
<td>maxPaletteIndex</td>
<td>7</td>
</tr>
<tr>
<td>maxPaletteColor</td>
<td>7</td>
</tr>
</tbody>
</table>

Modes

<table>
<thead>
<tr>
<th>djetBlack</th>
<th>0     default</th>
</tr>
</thead>
<tbody>
<tr>
<td>djet2Col</td>
<td>1     DeskJet 500C/550C</td>
</tr>
<tr>
<td>djet4Col</td>
<td>2     DeskJet 500C/550C</td>
</tr>
<tr>
<td>djet8Col</td>
<td>3     DeskJet 500C/550C</td>
</tr>
</tbody>
</table>

Printer modes are set using **SetDeviceMode**. The selection must be carried out *before* activating the device with **SetDevice**.
Error Codes

invalidCol 10002
invalidRow 10003

Specific error codes in MGDJet.

Colors

Data Types

DJetPaletteColor = Word;
Palette colors for DeskJet.

DJetPaletteArray = ARRAY[0..maxPaletteIndex] OF DJetPaletteColor;

DJetPaletteStruc = RECORD
  strucSize: CARDINAL;
  data : DJetPaletteArray;
END;

DJetPalette = ^DJetPaletteStruc;

The DevicePalette structure for DeskJet and compatible color printers can be used when calling MG.SetDevicePalette.
As a default the following palettes are used:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Device Color</th>
<th>djetBlack</th>
<th>djet2Col</th>
<th>djet4Col</th>
<th>djet8Col</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>black</td>
<td>black</td>
<td>black</td>
<td>black</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>white</td>
<td>white</td>
<td>blue</td>
<td>blue</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>green</td>
<td>green</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>white</td>
<td>cyan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>red</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>magenta</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>yellow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>white</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The only possible palette colors in the djetBlack mode are black and white (or the color of the paper). Using modes djet2Col..djet8Col the palette colors can be selected from among a set of fixed colors, namely black, blue, green, cyan, red, magenta, yellow and white (color of paper).

SetDeviceCall Functions

The Output Area - setSize

Constants

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>legal</td>
<td>0</td>
</tr>
<tr>
<td>letter</td>
<td>1</td>
</tr>
<tr>
<td>A4</td>
<td>2</td>
</tr>
<tr>
<td>maxSize</td>
<td>3</td>
</tr>
</tbody>
</table>

Default value: A4 (European format).

This specifies the sizes of paper that are supported. The constants are used when calling MG.SetDeviceCall. Here is how to set the US letter format:

```pascal
MG.SetDeviceCall(pr, MG.setSize, MGDJet.letter);
```
Data Types

\texttt{PrintSizeSet} = \texttt{SET OF 0..maxSize};

Dot Density - \texttt{setDensity}

Constants

\begin{align*}
\texttt{dens75} & \quad 0 \\
\texttt{dens100} & \quad 1 \\
\texttt{dens150} & \quad 2 \\
\texttt{dens300} & \quad 3 \\
\texttt{maxDens} & \quad 3
\end{align*}

Default value: \texttt{dens75}.

The density is specified in dots per inch (dpi). The constants are used when calling \texttt{MG.SetDeviceCall}. Here is how to set the density to 300 dpi:

\texttt{MG.SetDeviceCall(pr,MG.setDensity,MGDJet.dens300)};

Data Types

\texttt{DensitySet} = \texttt{SET OF 0..maxDens};

Data Compression - \texttt{MGDJet.setCompression}

Constants

\begin{align*}
\texttt{setCompression} & \quad 20
\end{align*}

Used as a parameter to \texttt{MG.SetDeviceCall} to select transmission mode to the printer (compressed or uncompressed).

\begin{align*}
\texttt{noCompression} & \quad 0 \\
\texttt{runLengthCompression} & \quad 1 \\
\texttt{maxCompression} & \quad 1
\end{align*}
Drivers D13-12  MGDJet - HP DeskJet Printers

Default value: **noCompression**.

This function is used to select the transmission mode from the computer to the printer. Compressed run-length encoding gives faster output, especially if the graphics contain large mono colored areas. The default setup is with compression disabled.

**Data Type**

\[ \text{CompressionSet} = \text{SET OF} \ 0..\text{maxCompression}; \]

**Color Intensity - MGDJet.setDepletion**

\[ \text{setDepletion} \quad 21 \]

Used as a parameter to `MG.SetDeviceCall` to select the level of color saturation at printout. The saturation is lowered by removing pixels (depletion) from the printout. This will sometimes enhance the quality. The command has only effect with color cartridges and the DeskJet 500C/550C printer.

- **noDepletion** 0
- **mediumDepletion** 1
- **maxDepletion** 2

Default value: **mediumDepletion** with the modes **djet2Col..djet8Col**.

**Data Type**

\[ \text{DepletionSet} = \text{SET OF} \ 0..\text{maxDepletion}; \]

**Multiple Passes - MGDJet.setShingling**

**Constants**

\[ \text{setShingling} \quad 22 \]

Used as a parameter to `MG.SetDeviceCall` to select the number of printing passes to reduce the risk of color bleeding when printing complex color
images. Shingling uses a checkerboard-pattern to print the graphics in multiple passes to let the ink dry before printing nearby pixels. The command has only effect with the DeskJet 500C/550C.

\[
\begin{align*}
&\text{noShingling} & 0 \\
&\text{mediumShingling} & 1 \\
&\text{maxShingling} & 2 \\
\end{align*}
\]

Default value: noShingling.

Data Type

\[
\text{ShinglingSet} = \text{SET OF 0..maxShingling};
\]
DeviceCall Functions

Mode Information - getDeviceModeInfo

Data Types

```pascal
DeviceModeInfo = RECORD
  sizes              : PrintSizeSet;
  defaultSize        : Word;
  densities          : DensitySet;
  defaultDens        : Word;
  depletions         : DepletionSet;
  defaultDepletion   : Word;
  shinglings         : ShinglingSet;
  defaultShingling   : Word;
  compressions       : CompressionSet;
  defaultCompression : Word;
  maxColorPlane      : Word;
  maxDeviceColor     : DeviceColor;
  maxColorMask       : DeviceColor;
  maxPaletteIndex    : Word;
  maxPaletteColor    : Word;
  defaultPal         : DJetPaletteStruc;
  rgbPal             : DJetRGBArray;
END;
```

This summarizes the capabilities and the default settings for a specific device mode (for example `djet8Col`). The settings can be read by calling `MG.DeviceCall`, function `MG.getDeviceModeInfo`. 
VAR
  pr        : MG.Device;
  prModeInfo: MGDJet.DeviceModeInfo;

MG.DeviceCall(pr,
  MG.getDeviceModeInfo,
  SizeOf(prModeInfo),
  prModeInfo);
IF MGDJet.dens300 IN prModeInfo.densities THEN BEGIN
  MG.SetDeviceCall(pr,
    MG.setDensity,
    MGDJet.dens300);
 ..

Current Mode - getDeviceSettings

Data Types

DeviceSettings = RECORD
  size        : Word;
  dens        : Word;
  depletion   : Word;
  shingling   : Word;
  compression : Word;
  width       : INTEGER;
  height      : INTEGER;
END;

This describes the current settings of a particular printer mode. Current settings can be read by calling \texttt{MG.DeviceCall}, function \texttt{MG.getDeviceSettings}.

VAR
  pr        : MG.Device;
  prSettings: MGDJet.DeviceSettings;

MG.DeviceCall(pr,
  MG.getDeviceSettings,
  SizeOf(prSettings),
  prSettings);
IF prSettings.dens = MGDJet.dens300 THEN ..
The graphics driver MGCanon (CANON.MGV, CANON.MGA/ CANON.XGA) works with Canon Laser LBP 4 - LBP 8.

MGCanon supports several different paper formats, resolutions from 75 dots/inch up to 300 dots/inch, etc. The default installation has been chosen to work with older printer models. N.B. printout routines and printer modes must be chosen before activating the printer by using MG.SetDevice:

MG.LoadDevice(pr,'CANON.MGA'); (*real mode*)
MG.SetDeviceOutput (pr,MGOut.LPT1);
MG.SetDeviceMode(pr,MGCanon...);
MG.SetDeviceCall(pr, MG.., MGCanon..);
...
MG.SetDevice(pr);
MG.SetBlackOnWhite(TRUE)..

In the example above, the printout is done via LPT1 (the parallel port). Remember that the laser printer is a "black-on-white" device!
Screens

The Visible Screen

Driver: CANON.MGV or CANON.MGA/CANON.XGA

The visible screen comprises the paper in the printer. The orientation can be set to portrait mode (default) as well as landscape mode. If the orientation is portrait, the paper's lower left corner has the position \( x = 0, y = 0 \). If the orientation is landscape, the paper's upper left corner has the position \( x = 0, y = 0 \). The sole drawing operation permitted is copying from the virtual and mono virtual screen respectively to the visible screen.

This means that all graphics must be drawn on the virtual or mono virtual screen so that afterwards it can be written out by copying to the visible screen using \texttt{MG.CopyView} or \texttt{MG.CopyRect}. Since images (\texttt{MGImg.Image}) are implemented using virtual screens, graphics can also be written out directly from images by calling \texttt{MGImg.DrawImage} or \texttt{MGImg.DrawImagePart}.

The only drawing mode permitted is replace. Positioning can to be done freely on the drawing area, which means that screens or images can be copied to any position on the paper before the page is printed. Of course, you can you create a large virtual screen that contains a full page and then copy the whole screen in one go, but this will increase the memory demands.

\textit{Before} activating the printer with \texttt{MG.SetDevice}, a printing routine must be installed. This is done by using \texttt{MG.SetDeviceOutput}. If you want to change the default portrait orientation you must do this before calling \texttt{MG.SetDevice}.

\texttt{MG.FlushVisiScr} is used to advance to a new page.

Mono virtual screens, and in certain cases virtual screens, can be in different graphics devices, provided that these have the same screen structure (1 bit/pixel) as Canon. In such cases, you can load CANON.MGV and save some memory.
Virtual Screens

Driver: CANON.MGA/CANON.XGA

All drawing modes and drawing operations can be used (1 bit/pixel). Copying to the visible screen is allowed, but not copying from the visible screen. Copying between virtual screens and mono virtual screens can also be done.

Memory requirements: A full A4 page, printed with the maximum resolution (300 dpi), requires approximately 1 MB memory! Instead of printing a full page at one go, you can use a smaller virtual screen and copy it several times to different positions on the same page. Furthermore, virtual screens can be allocated in virtual memory by including AutoVM in the USES list.

Mono Virtual Screens

The same data as for virtual screens.

Special Screens

Not available.

Line Style

Line styles in virtual and mono virtual screens are represented internally as values of 16-bits (2 bytes) according to the data type MGPat.LineStyle16Struc. See "MGPat - Standard Patterns " for examples on how to create your own line styles.

CanonLineStyle = ^MGPat.LineStyle16Struc;

Generic line styles in MGPat can be used.

Pattern

Patterns in virtual and mono virtual screens are represented internally as a rectangular bit pattern where the width is an even multiple of 8 (i.e. always complete bytes). The height can be one or more lines.
PatternStruc = RECORD
  strucSize: Word; (* = 6 + patSize *)
  byteWidth: Word;
  height : Word;
  pat     : ARRAY[1..patSize] OF Byte;
END;

CanonPattern = ^PatternStruc;

The size of the field pat is not fixed.

Bits set to "1" are drawn using the current foreground color while those set to "0" are drawn with the background color.

Most often 8x8 bit patterns of the internal type MGPat.Pattern8Struc are used at the lower print densities and 16x16 bit patterns of the internal type MGPat.Pattern16Struc at the higher resolutions. See MGPat - Standard Patterns" for declarations and examples on user defined patterns.

Generic patterns in MGPat can be used.

Printer FontsMGCanon can be used with bitmapped, stroke, outline and Bézier fonts.

Printout Outline

The outline below shows how to draw with full resolution on the laser printer. In "MG - Copy Graphics", you will also find an example of how to make a hard copy of the monitor screen.

USES MG, MGCanon, AutoVM;

VAR
  pr : MG.Device;
  prScr: MG.Screen;
  prWin: MG.Window;
 ..
MG.LoadDevice(pr,'CANON.MGA'); (*real mode*)
MG.SetDeviceOutput (pr,MGOut.LPT1);

(* Choose the printer orientation and configuration *)
MG.SetOrientation(pr,..);
Program Examples

**GAHCopy2** makes a hardcopy of the monitor screen to the printer. It works with all printer modes.

**GACanon** shows how graphics can be drawn on virtual screens and then printed. The program also shows how images of the type `MGImg.Image` can be handled directly and be printed using `MGImg.DrawImage`. It also shows how the monitor screen can be "dumped" to the printer. The program can be compiled with various resolutions.

**PCX Images**

`MGImg` can read and write monochrome PCX images.
Family Data

deviceFamily    11100
deviceType      rasterDevice
bitsPerPixel     1
maxDeviceColor   1
maxColorMask     1
maxColorPlane    0
maxPaletteIndex  1
maxPaletteColor  1
aspect           1000

Modes

canonISO        0
maxDeviceMode   0

MGCanon has only one mode, mode 0.

Error Codes

invalidCol      10002
invalidRow      10003
Colors

MGCanon supports only monochrome printers.

Data Types

CanonPaletteColor = 0..maxPaletteColor;

CanonPaletteArray =
  ARRAY[0..maxPaletteIndex] OF CanonPaletteColor;

CanonPaletteStruc = RECORD
  strucSize: Word;
  data     : CanonPaletteArray;
END;

CanonPalette = ^CanonPaletteStruc;

The DevicePalette structure for Canon Laser and compatible printers can be used when calling MG.SetDevicePalette.

SetDeviceCall Functions

The Output Area - setSize

Constants

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>legal</td>
<td>0 8.5&quot; x 14&quot;</td>
</tr>
<tr>
<td>letter</td>
<td>1 8.5&quot; x 11&quot;</td>
</tr>
<tr>
<td>A4</td>
<td>2 210 x 297 mm</td>
</tr>
<tr>
<td>B5</td>
<td>3 182 x 257 mm</td>
</tr>
<tr>
<td>maxSize</td>
<td>3</td>
</tr>
</tbody>
</table>

Default value: A4 (European/international format).
This specifies the sizes of paper that are supported. The constants are used when calling `MG.SetDeviceCall`. Here is how you set the US letter format:

```pascal
MG.SetDeviceCall(pr, MG.setSize, MGCanon.letter);
```

The paper orientation (portrait or landscape) is set by calling `MG.SetOrientation`.

Data Types

```pascal
PrintSizeSet = SET OF 0..maxSize;
```

Dot Density - `setDensity`

Constants

```pascal
dens75    0    75 dots/inch
dens100   1    100 dots/inch
dens150   2    150 dots/inch
dens300   3    300 dots/inch
maxDens   3
```

Default value: `dens75`.

The density is specified in dots per inch or `dpi` (the same resolution horizontally and vertically). The constants are used when calling `MG.SetDeviceCall`.

The greater the resolution, the longer the printout will take. Since the printing time increases faster than the resolution, it is often worthwhile to do test printouts using lower resolutions and to save the full resolution until the final version is to be printed.

Here is how you set the resolution to 300 dots/inch:

```pascal
MG.SetDeviceCall(pr, MG.setDensity, MGCanon.dens300);
```

Data Types

```pascal
DensitySet = SET OF 0..maxDens;
```
DeviceCall Functions

Mode Information - getDeviceModeInfo

Data Types

\[
\text{DeviceModeInfo} = \text{RECORD}
\begin{array}{l}
\text{sizes} & : \text{PrintSizeSet}; \\
\text{defaultSize} & : \text{Word}; \\
\text{densities} & : \text{DensitySet}; \\
\text{defaultDens} & : \text{Word}; \\
\text{charSets} & : \text{SetOfCharacterSet}; \\
\text{defaultCharSet} & : \text{Word};
\end{array}
\text{END;}
\]

This summarizes the capabilities and the default settings for Canon Laser. The settings can be read by calling \text{MG.DeviceCall}, function \text{MG.getDeviceModeInfo}.

\[
\text{VAR}
\begin{array}{l}
\text{pr} & : \text{MG.Device}; \\
\text{prModeInfo} & : \text{MGCanon.DeviceModeInfo};
\end{array}
\]

\text{MG.DeviceCall(pr,}
\begin{array}{l}
\text{MG.getDeviceModeInfo,} \\
\text{SizeOf(prModeInfo),} \\
\text{prModeInfo)};
\end{array}
Current Mode - getDeviceSettings

Data Types

\[
\text{DeviceSettings} = \text{RECORD} \\
\quad \text{size} : \text{Word}; \\
\quad \text{dens} : \text{Word}; \\
\quad \text{width} : \text{INTEGER}; \\
\quad \text{height} : \text{INTEGER}; \\
\quad \text{charSet} : \text{Word}; \\
\text{END}; \\
\]

This describes the current settings of a particular printer mode. Current settings can be read by calling \text{MG.DeviceCall}, function \text{MG.getDeviceSettings}.

VAR
\[
\text{pr} : \text{MG.Device}; \\
\text{prSettings} : \text{MGCanon.DeviceSettings}; \\
\]

\text{MG.DeviceCall}(\text{pr}, \\
\quad \text{MG.getDeviceSettings}, \\
\quad \text{SizeOf(\text{prSettings})}, \\
\quad \text{prSettings}); \\
\text{IF prSettings.dens = MGCanon.dens300 THEN .}
Appendix
A P P E N D I X  A

Error Messages

This appendix contains a summary of all error codes that you may encounter when calling procedures in MultiGraphics. The codes can be displayed as decimals or hexadecimal numbers by error procedures (normally directly on screen) or may be checked by calling SV.Result if the use of error procedures is set to off (SV.SetEP(FALSE). See "MG - Error Handling" for more information on the SuperVisor error handling technique in MultiGraphics.

SVErrMsg

By importing the unit SVErrMsg, the global output procedure for error messages (SV.GlbEP) is modified to include error messages in plain English. The text messages are loaded from the file "SVERRMSG.TXT" which must be located in the current directory. This is a text file with one error message per line. The decimal error code consists of the first 5 positions at the beginning of each line, followed by a space and then the error message string.

USES SVErrMsg, MG, MGImg;
## Error Messages

### SV

<table>
<thead>
<tr>
<th>Decimal Error</th>
<th>Symbolic Error</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ok</td>
<td>OK!</td>
</tr>
<tr>
<td>1</td>
<td>noMem</td>
<td>Out of memory for termination procedures</td>
</tr>
</tbody>
</table>

### I/O errors

For I/O errors such as read or write errors to disk, wrong filename, wrong directory, or disk full etc., Turbo Pascal's I/O routines are returned with a negative sign!

<table>
<thead>
<tr>
<th>Decimal Error</th>
<th>Symbolic Error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>-</td>
<td>I/O error</td>
</tr>
</tbody>
</table>

### MGErr

This unit declares a number of error codes that are used by the unit MG.

<table>
<thead>
<tr>
<th>Decimal Error</th>
<th>Symbolic Error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>notImplemented</td>
<td>this function is not implemented in the driver</td>
</tr>
<tr>
<td>1001</td>
<td>processWithoutWin</td>
<td>the calling process has no window</td>
</tr>
<tr>
<td>1002</td>
<td>outOfMem</td>
<td>not enough memory</td>
</tr>
<tr>
<td>1003</td>
<td>notScrWin</td>
<td>the current window must be a screen</td>
</tr>
<tr>
<td></td>
<td>onlyInScrWin</td>
<td>only allowed from the screen window</td>
</tr>
<tr>
<td>1004</td>
<td>invalidWinSpec</td>
<td>invalid window data</td>
</tr>
<tr>
<td>1005</td>
<td>deviceInUse</td>
<td>device already open</td>
</tr>
<tr>
<td>1006</td>
<td>failOpenVisiScreen</td>
<td>the visible screen could not be opened</td>
</tr>
</tbody>
</table>
## Error Messages

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1007</td>
<td><code>driverNotFound</code> cannot find the driver file</td>
</tr>
<tr>
<td>1009</td>
<td><code>invalidScreenType</code> invalid screen type</td>
</tr>
<tr>
<td>1010</td>
<td><code>invalidWinSize</code> invalid window width and/or height</td>
</tr>
<tr>
<td>1011</td>
<td><code>notAllowedInScreenWin</code> call not allowed in screen window</td>
</tr>
<tr>
<td>1012</td>
<td><code>invalidViewSpec</code> invalid data for the viewport</td>
</tr>
<tr>
<td>1013</td>
<td><code>invalidWin</code> invalid window variable</td>
</tr>
<tr>
<td>1014</td>
<td><code>invalidDeviceColor</code> invalid device color</td>
</tr>
<tr>
<td>1015</td>
<td><code>invalidDrawMode</code> invalid drawing mode</td>
</tr>
<tr>
<td>1016</td>
<td><code>invalidLineWidth</code> invalid line width</td>
</tr>
<tr>
<td>1017</td>
<td><code>invalidPattern</code> invalid pattern</td>
</tr>
<tr>
<td>1018</td>
<td><code>invalidDriver</code> invalid driver (wrong version)</td>
</tr>
<tr>
<td>1020</td>
<td><code>invalidAspect</code> invalid aspect</td>
</tr>
<tr>
<td>1021</td>
<td><code>invalidScreen</code> invalid screen variable</td>
</tr>
<tr>
<td>1022</td>
<td><code>invalidWinChain</code> internal chain of windows damaged</td>
</tr>
<tr>
<td>1023</td>
<td><code>invalidScreenChain</code> internal chain of screens damaged</td>
</tr>
<tr>
<td>1024</td>
<td><code>driverNotLoaded</code> driver not loaded</td>
</tr>
<tr>
<td>1025</td>
<td><code>invalidRadiusOrAxis</code> invalid radius or axis</td>
</tr>
<tr>
<td>1026</td>
<td><code>tooManyPoints</code> too many polygon points</td>
</tr>
<tr>
<td>1027</td>
<td><code>alreadyLinked</code> capability already linked</td>
</tr>
<tr>
<td>1028</td>
<td><code>capDontExist</code> capability not installed</td>
</tr>
<tr>
<td>1029</td>
<td><code>capInitError</code> error when initiating a linked capability</td>
</tr>
<tr>
<td>1030</td>
<td><code>invalidFont</code> invalid font variable</td>
</tr>
<tr>
<td>1031</td>
<td><code>invalidSymbols</code> invalid symbol variable</td>
</tr>
<tr>
<td>1032</td>
<td><code>invalidSymbolsFile</code> invalid symbol file</td>
</tr>
<tr>
<td>1033</td>
<td><code>invalidFontFile</code> invalid font file</td>
</tr>
<tr>
<td>1034</td>
<td><code>invalidTransform</code> invalid symbol or font transformation</td>
</tr>
<tr>
<td>1035</td>
<td><code>invalidAngle</code> invalid angle</td>
</tr>
<tr>
<td>1037</td>
<td><code>invalidRGB</code> invalid RGB value</td>
</tr>
<tr>
<td>1038</td>
<td><code>outsideView</code> the current position is outside the viewport</td>
</tr>
<tr>
<td>1039</td>
<td><code>notValidInDevice</code> not implemented in device driver</td>
</tr>
<tr>
<td>1040</td>
<td><code>deviceProcNotInUse</code> the driver does not support the procedure</td>
</tr>
<tr>
<td>1041</td>
<td><code>capTermError</code> error in closing window capability</td>
</tr>
<tr>
<td>1042</td>
<td><code>invalidConfig</code> invalid configuration</td>
</tr>
<tr>
<td>1043</td>
<td><code>invalidSymbolsType</code> invalid symbol type</td>
</tr>
<tr>
<td>1044</td>
<td><code>invalidFontType</code> invalid font</td>
</tr>
<tr>
<td>1045</td>
<td><code>symbolToComplex</code> too many points in symbol</td>
</tr>
<tr>
<td>1046</td>
<td><code>onlyInVisiScreenWin</code> only allowed in the visible screen's window</td>
</tr>
<tr>
<td>1047</td>
<td><code>lockError</code> lock error with virtual memory</td>
</tr>
<tr>
<td>1048</td>
<td><code>unlockError</code> unlock error with virtual memory</td>
</tr>
<tr>
<td>1049</td>
<td><code>invalidSymbolStr</code> invalid text string with symbol set</td>
</tr>
<tr>
<td>1051</td>
<td><code>incompatibleDevices</code> non-compatible copying devices</td>
</tr>
</tbody>
</table>
Appendix A-4  

Error Messages

1052  invalidMaxColorMask  invalid max. color mask
1053  invalidColorMask  invalid color mask
1054  invalidPalette  invalid palette
1055  invalidRGBPalette  invalid RGB-palette
1056  invalidPaletteDest  invalid destination when copying palettes
1057  invalidLineStyle  invalid line style
1058  invalidPolygon  invalid polygon specification
1059  invalidLineStylePos  invalid line style position

1060  invalidParameter  invalid parameter to DeviceCall
1061  deviceNotFound  device not found or is missing
1062  invalidDeviceFont  invalid device font
1063  invalidOrientation  invalid orientation
1064  invalidDeviceMode  invalid device mode
1065  outOfWindowHandles  no more window handles
1066  symbolOutOfRange  invalid symbol number
1067  invalidIndex  invalid index when outline tracing
1068  invalidStartPoint  invalid starting point when outline tracing
1069  invalidDisplayList  invalid list of drawing commands
1070  displayProcOutOfRange  invalid number for drawing procedure
1071  invalidDisplayProc  invalid drawing procedure
1072  displayArrayFull  internal array of linked playback units is full

1075  bufferBusy  MG.bufferPtr is busy
1076  invalidBlockSize  invalid bit block size
1077  invalidBlock  invalid bit block
1078  invalidPointer  invalid pointer
1079  invalidSymbolData  invalid symbol data
1080  invalidSymbolWidth  invalid symbol width
1081  invalidSymbolHeight  invalid symbol height
1082  invalidHotSpot  invalid symbol hot spot
1083  invalidBaseline  invalid character base line
1084  symbolTooLarge  symbol data too large
## Error Messages

### MGW

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1103</td>
<td>invalidWorld</td>
<td>invalid world coordinates</td>
</tr>
<tr>
<td>1104</td>
<td>invalidPos</td>
<td>invalid world position</td>
</tr>
<tr>
<td>1105</td>
<td>invalidAngle</td>
<td>invalid world angle</td>
</tr>
<tr>
<td>1106</td>
<td>invalidRect</td>
<td>invalid rectangle (FillRect)</td>
</tr>
</tbody>
</table>

### MGIW

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>invalidWorld</td>
<td>invalid world coordinates</td>
</tr>
<tr>
<td>1124</td>
<td>invalidPos</td>
<td>invalid world position</td>
</tr>
<tr>
<td>1126</td>
<td>invalidRec</td>
<td>invalid rectangle (MGIW.FillRect)</td>
</tr>
</tbody>
</table>

### MGEv

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1150</td>
<td>eventHandlerActive</td>
<td>event handler already activated</td>
</tr>
<tr>
<td>1151</td>
<td>eventHandlerNotActive</td>
<td>event handler not activated</td>
</tr>
<tr>
<td>1152</td>
<td>invalidServerProc</td>
<td>invalid event service procedure</td>
</tr>
<tr>
<td>1153</td>
<td>invalidEventGenerator</td>
<td>invalid event generator</td>
</tr>
</tbody>
</table>

### MGClock

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1160</td>
<td>clockActive</td>
<td>the clock's event generating is already active</td>
</tr>
<tr>
<td>1161</td>
<td>invalidAlarm</td>
<td>invalid alarm number</td>
</tr>
<tr>
<td>1162</td>
<td>invalidTicks</td>
<td>the number of ticks is too large</td>
</tr>
<tr>
<td>1163</td>
<td>invalidTime</td>
<td>invalid time given</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(hours, minutes, seconds)</td>
</tr>
</tbody>
</table>
### MGKbd

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Symbolic</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1170</td>
<td>kbdActive</td>
<td>keyboard event generating already active</td>
</tr>
<tr>
<td>1171</td>
<td>kbdNotActive</td>
<td>keyboard event generating not active</td>
</tr>
</tbody>
</table>

### MGMouse

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Symbolic</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1180</td>
<td>invalidMouse</td>
<td>mouse not activated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(mouse driver missing,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StartMouseEvents not called)</td>
</tr>
<tr>
<td>1181</td>
<td>invalidPos</td>
<td>mouse position outside screen</td>
</tr>
<tr>
<td>1182</td>
<td>invalidPointer</td>
<td>invalid mouse pointer</td>
</tr>
<tr>
<td>1183</td>
<td>invalidSens</td>
<td>invalid sensitivity value</td>
</tr>
<tr>
<td>1184</td>
<td>mouseHandlerActive</td>
<td>Emulation mode can only be changed before StartMouseEvents</td>
</tr>
</tbody>
</table>

### MGIO

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Symbolic</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>invalidCursor</td>
<td>the cursor is not in the current font</td>
</tr>
<tr>
<td>1201</td>
<td>maxLengthError</td>
<td>the position is greater than the actual string length</td>
</tr>
<tr>
<td>1202</td>
<td>invalidInteger</td>
<td>invalid integer</td>
</tr>
<tr>
<td>1203</td>
<td>invalidScrollLines</td>
<td>the number of scroll lines must be ( \geq 0 )</td>
</tr>
<tr>
<td>1204</td>
<td>invalidTermKeySet</td>
<td>an empty set is not allowed</td>
</tr>
<tr>
<td>1205</td>
<td>invalidInputProc</td>
<td>invalid input procedure</td>
</tr>
<tr>
<td>1206</td>
<td>invalidWord</td>
<td>invalid integer</td>
</tr>
<tr>
<td>1207</td>
<td>invalidLongInt</td>
<td>invalid long integer</td>
</tr>
<tr>
<td>1208</td>
<td>invalidReal</td>
<td>invalid floating-point number</td>
</tr>
<tr>
<td>1209</td>
<td>invalidDrawCharProc</td>
<td>invalid character drawing procedure</td>
</tr>
<tr>
<td>1215</td>
<td>invalidAttributes</td>
<td>invalid character attributes</td>
</tr>
</tbody>
</table>
**Error Messages**

### MGCol

<table>
<thead>
<tr>
<th>Decimal Error</th>
<th>Symbolic Error</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1250</td>
<td>invalidHSV</td>
<td>invalid value of hue, sat or value</td>
</tr>
</tbody>
</table>

### MGIImg

<table>
<thead>
<tr>
<th>Decimal Error</th>
<th>Symbolic Error</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1301</td>
<td>invalidImage</td>
<td>invalid image variable</td>
</tr>
<tr>
<td>1302</td>
<td>invalidImageScreen</td>
<td>invalid virtual image screen</td>
</tr>
<tr>
<td>1303</td>
<td>invalidMaskScreen</td>
<td>invalid monochrome screen mask</td>
</tr>
<tr>
<td>1304</td>
<td>invisibleImage</td>
<td>the image is outside the current screen</td>
</tr>
<tr>
<td>1305</td>
<td>noScreen</td>
<td>the virtual screen is missing</td>
</tr>
<tr>
<td>1306</td>
<td>invalidSkips</td>
<td>the mask does not have the same indent as the image (AddImageMask)</td>
</tr>
<tr>
<td>1307</td>
<td>invalidImagePart</td>
<td>invalid parameter in (DrawImagePart)</td>
</tr>
<tr>
<td>1308</td>
<td>invalidImageFile</td>
<td>invalid image file</td>
</tr>
<tr>
<td>1309</td>
<td>invalidPCXfile</td>
<td>invalid PCX/PCC file</td>
</tr>
<tr>
<td>1310</td>
<td>readBlockError</td>
<td>internal read error in generating PCX file</td>
</tr>
</tbody>
</table>

### MGFnt

<table>
<thead>
<tr>
<th>Decimal Error</th>
<th>Symbolic Error</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1350</td>
<td>invalidBGIFile</td>
<td>invalid BGI font file</td>
</tr>
</tbody>
</table>
### MGOut

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1425</td>
<td>invalidCOMData</td>
<td>invalid parameters when initializing the serial port</td>
</tr>
<tr>
<td>1426</td>
<td>initCOMError</td>
<td>I/O error when initializing the serial port</td>
</tr>
<tr>
<td>1427</td>
<td>diskInUse</td>
<td>DISK device already active and accessed by another device</td>
</tr>
<tr>
<td>1428</td>
<td>diskNotInUse</td>
<td>DISK device not activated</td>
</tr>
<tr>
<td>1429</td>
<td>invalidDeviceOutput</td>
<td>invalid output routine</td>
</tr>
</tbody>
</table>

### MGDL

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500</td>
<td>invalidDisplayListFile</td>
<td>Invalid display list file</td>
</tr>
</tbody>
</table>

### MGDO

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1525</td>
<td>invalidDisplayObjects</td>
<td>Invalid display object library</td>
</tr>
<tr>
<td>1526</td>
<td>displayObjectsUsed</td>
<td>Display objects used</td>
</tr>
<tr>
<td>1527</td>
<td>invalidDisplayObjectId</td>
<td>Invalid display object handle</td>
</tr>
<tr>
<td>1528</td>
<td>usedDisplayObjectId</td>
<td>Display object handle in use</td>
</tr>
<tr>
<td>1529</td>
<td>invalidDisplayObjectsFile</td>
<td>Invalid display object file</td>
</tr>
<tr>
<td>1530</td>
<td>invalidDisplayObjectNames</td>
<td>Invalid display object names</td>
</tr>
</tbody>
</table>
### VM

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>131</td>
<td>invalidHandle</td>
<td>invalid memory block parameter</td>
</tr>
<tr>
<td>135</td>
<td>outOfMem</td>
<td>not enough memory</td>
</tr>
<tr>
<td>2200</td>
<td>alreadyLocked</td>
<td>memory block already locked</td>
</tr>
<tr>
<td>2201</td>
<td>notLocked</td>
<td>memory block not locked</td>
</tr>
<tr>
<td>2202</td>
<td>invalidSize</td>
<td>invalid size of memory block</td>
</tr>
<tr>
<td>2203</td>
<td>invalidOffset</td>
<td>invalid offset in memory block</td>
</tr>
<tr>
<td>2204</td>
<td>invalidMemManager</td>
<td>invalid virtual memory manager</td>
</tr>
<tr>
<td>2205</td>
<td>lockedHandle</td>
<td>memory block locked</td>
</tr>
<tr>
<td>2206</td>
<td>invalidLockSize</td>
<td>invalid lock size</td>
</tr>
</tbody>
</table>

### DBList

<table>
<thead>
<tr>
<th>Decimal error</th>
<th>Symbolic error</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2300</td>
<td>notAllocated</td>
<td>List not allocated</td>
</tr>
<tr>
<td>2301</td>
<td>outOfSpace</td>
<td>Out of memory</td>
</tr>
<tr>
<td>2302</td>
<td>emptyList</td>
<td>The list is empty</td>
</tr>
<tr>
<td>2303</td>
<td>invalidPos</td>
<td>Invalid list position</td>
</tr>
<tr>
<td>2304</td>
<td>paraLTData</td>
<td>Parameter data size &lt; list element size</td>
</tr>
<tr>
<td>2305</td>
<td>paraGTData</td>
<td>Parameter data size &gt; list element size</td>
</tr>
<tr>
<td>2306</td>
<td>invalidListFile</td>
<td>Invalid list file</td>
</tr>
<tr>
<td>2307</td>
<td>invalidElements</td>
<td>Invalid number of elements</td>
</tr>
</tbody>
</table>
Computer graphics require a lot of memory. The memory requirements for fonts, symbols, driver routines and to an even greater degree, images and virtual screens, can be measured in tens, hundreds or even thousands of kilobytes. Using the DOS environment's infamous 640 KB limit for the so-called *conventional memory*, it does not take long before memory resources are used up. Applications, such as creating images as large as that of a SuperVGA screen or a page for laser printout using 600 dots/inch, are practically impossible to handle.

The best and most effortless solution is to simply recompile the application for *protected/extended mode*. However, in some situations it is necessary to produce applications for *real mode* and *MultiGraphics* is therefore based on a technique for *virtual memory management*, which pulls down the walls DOS has set up and makes it possible to utilize the "unlimited" memory resources that are available using extended XMS memory ("eXtended Memory Specification"), expanded EMS memory ("Expanded Memory Specification") and hard disks.

Virtual memory can be used for fonts, symbols, virtual screens and images. All drawing commands work exactly the same, no matter where in the total virtual memory the objects are stored.
Virtual memory management is based on the unit **VM** coming with **MultiGraphics**. Using **VM**, memory can be allocated (reserved) on the conventional program "heap" in much the same way as with **New** and **GetMem**, but with the significant improvement that allocated memory blocks can be much bigger than 64 KB, and that they are not limited to a total of 640 KB.

With the additional units, **XMSVM**, **EmsVM** and **FileVM** virtual memory can be allocated in extended XMS memory, expanded EMS memory or as files on hard disks or RAM disks. By exploiting the unit **AutoVM**, the choice of the physical memory type can be managed automatically and, from the programmer's point of view, all available memory can be seen as a single, huge memory pool.

**XMSVM**

**XMSVM** requires extended memory and a loaded XMS driver, normally HIMEM.SYS in MS-DOS version 5.0 or higher, or alternatively a memory handler such as 386MAX/BLUEMAX or QEMM.

**EmsVM**

**EmsVM** functions with both EMS version 3.2 and 4.0. EMS memory (or "expanded memory") can be installed using special EMS memory cards or, on 386/486 machines, be emulated by software using extended memory. EMS emulation is included in MS-DOS version 5.0 or higher (EMM386) and in memory managers such as 386MAX/BLUEMAX and QEMM. EMS memory can be added up to 32 MB, which should be sufficient for now at least!

The difference in performance between drawing in XMS or EMS memory compared to conventional memory is almost negligible, especially in the case of EMS memory.

**FileVM**

**FileVM** let you use files on disk (hard disk or RAM disk) as a memory reserve pool. Performance when drawing is, of course, slower than when using conventional memory, XMS or EMS memory. Therefore, virtual memory on disk should only be used as a last resort or as an archive for seldom used images and virtual screens, etc.
AutoVM

The simplest way for a graphics application to get access to all available virtual memory is by adding AutoVM to the USES list of the program. That's all! This is how it looks:

USES
  MG, MGImg, SV, AutoVM;
VAR
  img: MGImg.Image;
 ..
MGImg.ReadImage(img,..);
IF SV.Result <> SV.ok THEN BEGIN
  (* Virtual memory is used up — error message *)
 ..
END;
MGImg.DrawImage(img);
 ..
MGImg.DisposeImage(img);

As you see, the only measure taken is that we have added AutoVM to the USES list!

Behind the scene, AutoVM builds on the units VM, EmsVM, XMSVM and FileVM, and a specific allocation strategy. When memory is requested, AutoVM makes a choice among the available memory resources in accordance with the following strategy:

1) If the requested memory block is <= 8092 bytes, conventional heap memory will be used if available. The VM module is used.

2) If the memory block is > 8092 bytes, or conventional memory is not available, AutoVM will try to use expanded EMS memory if available. The EmsVM unit is used.

3) If expanded memory is not available, AutoVM checks if extended XMS memory is available. The XMSVM unit is used.

4) If extended memory is not available and the block is > 8092 bytes, AutoVM will try to use conventional heap memory once again. The VM unit is used.

5) If AutoVM has failed up to this point it will try to store the block on disk. The FileVM unit is used.

6) If the disk is out of space, an error will be reported.
Virtual Memory Management

EMS memory is selected before XMS memory, because EMS is faster when moving huge data blocks (like images and virtual screens) between virtual memory and conventional memory, which is necessary when doing graphics operations on the data. This is called memory locking and unlocking.

If neither conventional memory nor XMS or EMS memory is available or already allocated, perhaps a disk can be used. You can specify the desired disk and directory for the virtual memory handling. This is how it's done:

```pascal
USES MG,AutoVM,FileVM,...;
...
FileVM.SetPathFVM('D:\temp');
...
MG.CreateMonoVirtScr(scr,5000,5000);
...
```

If you don't call `SetPathFVM`, the current disk and directory will be used. If the system has a fast RAM disk, performance is enhanced considerably. Even a good disk caching program speeds everything up.

When locking a memory block on disk, a buffer of 16 KB at the most is used. If two disk blocks are locked simultaneously, a maximum of 2 x 16 KB is needed. In order to avoid being "locked out" (usually the error `lockError`), you should see to it that there is always at least 16 KB of free memory on the program heap before allocating memory from the heap.

Manual Selection of Virtual Memory

The choice of the type of virtual memory can also be controlled "manually".

If, for example, you want a program to use EMS memory, the units `VM` and `EmsVM` must be specified and the virtual memory handler be informed that memory allocation in EMS is wanted from then on.
USES MG, MGImg, VM, EmsVM;

.. MG.LoadFont(fnt,..); (*fnt is loaded in conventional memory*)
MGImg.ReadImage(img1,..); (*img1 is placed in conventional memory*)
VM.SetVM(EmsVM.vmm); (*all new allocations in EMS*)
MGImg.ReadImage(img2,..); (*img2 is placed in EMS memory*)
.. MG.SetFont(fnt); (*fnt in conventional memory is used*)
MGImg.DrawImage(img2); (*img2 in EMS is drawn*)
MGImg.DrawImage(img1); (*img1 in conventional memory is drawn*)
..
MG.DisposeFont(fnt); (*fnt in conventional memory is deallocated*)
MGImg.DisposeImage(img1); (*img1 in conventional memory is deallocated*)
MGImg.DisposeImage(img2); (*img2 in EMS is deallocated*)
..

As the example shows, the use of conventional memory can be mixed with EMS memory (as can XMS memory and memory on disk). The current virtual memory for allocations is specified by VM.SetVM and this memory is used from then on when allocating new memory blocks, until SetVM is called once again. The virtual memory handler itself keeps track of where previously allocated memory blocks are.
# Appendix C

## Font Files

The following font files are included with **MultiGraphics**:

<table>
<thead>
<tr>
<th>Font file</th>
<th>Type</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS8X8.CBS</td>
<td>bitmap</td>
<td>fixed</td>
<td>standard system font</td>
</tr>
<tr>
<td>SYS8X14.CBS</td>
<td>bitmap</td>
<td>fixed</td>
<td>standard system font</td>
</tr>
<tr>
<td>SYS8X16.CBS</td>
<td>bitmap</td>
<td>fixed</td>
<td>standard system font</td>
</tr>
<tr>
<td>SYS8X19.CBS</td>
<td>bitmap</td>
<td>fixed</td>
<td>standard system font</td>
</tr>
<tr>
<td>SYS12X24.CBS</td>
<td>bitmap</td>
<td>fixed</td>
<td>standard system font</td>
</tr>
<tr>
<td>SYS12X20.CBS</td>
<td>bitmap</td>
<td>fixed</td>
<td>standard system font</td>
</tr>
<tr>
<td>FIXED10.CBS</td>
<td>bitmap</td>
<td>fixed</td>
<td>alternate system font</td>
</tr>
<tr>
<td>TYPEWR10.CBS</td>
<td>bitmap</td>
<td>fixed</td>
<td>&quot;typewriter&quot; style</td>
</tr>
<tr>
<td>SANS8.CBS</td>
<td>bitmap</td>
<td>prop</td>
<td>simple style for headings</td>
</tr>
<tr>
<td>SANS10.CBS</td>
<td>bitmap</td>
<td>prop</td>
<td>simple style for headings</td>
</tr>
<tr>
<td>SANS12.CBS</td>
<td>bitmap</td>
<td>prop</td>
<td>simple style for headings</td>
</tr>
<tr>
<td>ROMAN8.CBS</td>
<td>bitmap</td>
<td>prop</td>
<td>body text style</td>
</tr>
<tr>
<td>ROMAN10.CBS</td>
<td>bitmap</td>
<td>prop</td>
<td>body text style</td>
</tr>
<tr>
<td>ROMAN12.CBS</td>
<td>bitmap</td>
<td>prop</td>
<td>body text style</td>
</tr>
<tr>
<td>SANS.COL</td>
<td>outline</td>
<td>prop</td>
<td>simple scaleable style</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>for headings</td>
</tr>
<tr>
<td>ROMAN.COL</td>
<td>outline</td>
<td>prop</td>
<td>scaleable body text style</td>
</tr>
<tr>
<td>TWRITER.CZL</td>
<td>Bézier</td>
<td>fixed</td>
<td>scaleable &quot;typewriter&quot; style</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(unscaled height = 1024 pixels)</td>
</tr>
</tbody>
</table>
The fonts in SANS.COL, ROMAN.COL and TWRITER.CZL are very large when unscaled, but resist to scale them down too much if you want to avoid annoying round-off errors. These fonts are best adapted for high resolution screen modes, printers and plotters.

BGI compatible stroke fonts (’.CHR’) from Borland can be used with MGFnt.LoadBGI. See "MGFnt - Font Conversion" for all details.
The unit **MGBGI** emulates *Borland's Graphic Interface* (BGI) and makes it possible to use programs or routines written with Borland graphics together with **MultiGraphics**. There are advantages in using emulation both when working with plain BGI applications, as well as when mixing BGI code with **MultiGraphics** code.

In the latter case, we recommend of course converting BGI commands to **MultiGraphics** commands, since **MultiGraphics** 1) has a clearer design and 2) it is easier to maintain applications using one instead of two graphics interfaces. Still, when you're in hurry, it be may motivated to get a program up and running as quick as possible, instead of putting efforts into a conversion. The BGI emulation is implemented using standard **MultiGraphics** commands, and thanks to this, BGI and **MultiGraphics** commands can coexist within an application without any problems.

Even in pure BGI applications, that you do not intend to convert to **MultiGraphics** commands, there are in fact several advantages in exchanging Borland's standard unit to our emulation unit. Here are some improvements you will get when using **MGBGI**:

- Support for *more graphics devices* such as SuperVGA, complete 8514/A support, support for dot matrix, laser and ink jet printers and color plotters.

- Support for BGI graphics even on *virtual screens*.

**MultiGraphics & Borland Pascal/Turbo Pascal**

**DATABITEN**
Appendix D-2  MGBGI - Emulation of BGI

- Support for window oriented BGI graphics - each MultiGraphics window makes up a separate BGI world with its own repertoire of data for current position, color, text direction, etc. By changing windows, you will change to a new BGI world at the same time!

- Support for images > 64 KB (up to 16384x16384 pixels), which means that a complete VGA screen can be saved in one image.

- Support for 256-color drawing modes and improved palette handling.

- Support for virtual memory, which dramatically increases the available memory for virtual screens, images, fonts and symbols.

- Support for both BGI stroke fonts and MultiGraphics bitmapped, stroke, outline and Bézier fonts.

- Support for improved clipping of characters and images.

- Support for the drawing mode stamp, which makes it easy to draw objects with transparent parts.

- Support for symbol handling.

- Support for storing and reading of images in PCX format.

- Support for zooming of images,

- Support for advanced text handling and text i/o etc., using the unit MGIO. Standard Read, ReadLn, Write and WriteLn can be used in graphics mode as well.

- Support for improved error handling using SuperVisor.

- Support for event handling with the clock, keyboard and mouse.

MGBGI emulates, with some few exceptions, all BGI routines, constants and data types. Moreover, it includes further constants, data types, variables and procedures for among other things SuperVGA, etc.
Drivers and Font Files

During emulation, the graphics drivers in MultiGraphics are used - not Borland's BGI driver routines.

Both Borland's font files (".CHR") and MultiGraphics's own font files can be used.

Some Differences

The default font for VGA has a size of 8x8 points with the original BGI, while MultiGraphics uses 8x14 points (which looks better).

With emulation, only stroke, outline and Bézier fonts can be scaled, not the bitmapped default font.

There are certain differences in sizes when scaling fonts.

In the overview below, we only document inconsistencies compared to Borland's original unit (version 3.0).

Simple Adaptation of BGI Applications

In order to compile an existing BGI application using the emulation, you must first take a few measures:

· Search for "GetImage" and "PutImage" and then locate the declarations of the used image variables. Change the declarations to use the Image type. Search for "ImageSize" and remove the allocations of memory for image variables (that are carried out automatically by GetImage using emulation). Search for deallocations of image memory as well and replace these with DisposeImage.
Advanced Use of Emulation

BGI emulation can be combined with other routines in MultiGraphics, e.g., support for printers and plotters, handling of windows, virtual screens, color handling, image handling, event handling, text i/o, etc.

BGI as a Window Capability

BGI emulation is implemented as a window capability in MultiGraphics. When initializing a graphics routine using InitGraph, the visible screen's full screen window receives this BGI capability, as well as all the windows that hereafter are opened in the visible screen. If you create your own virtual screens, these must, however, be given the BGI capability manually (with MG.EnhanceWin and/or MG.EnhanceScr). Here is an example:

```
MG.CreateVirtScr(scr,400,200,15);
MG.SetScr(scr);
MG.EnhanceWin(MG.CurWin,MGBGI.cap);
MG.EnhanceScr(MG.CurScr,MGBGI.cap);
MG.CreateWin(win,100,50,200,100);
MG.SetWin(win);
OutTextXY(5,10,'Virtual screen with BGI');
```

The same (i.e. that the BGI capability must be added manually) applies if the device has not been opened using the BGI command InitGraph but with MG.AutoDevice, MG.LoadDevice or MG.OpenDevice.

Each window under MultiGraphics, that has the BGI capability, becomes its own BGI world with a separate series of BGI data. Changes to position, text style, fill pattern etc., in one window do not affect status for other windows.

Program Examples

The example GABGI shows simple direct emulation of the BGI as well as a selection of those new possibilities that are given thanks to the combination with MultiGraphics. There are several more examples on the program disk that is supplied with "Animation on the PC".
Constants

Constants not Available

The following graphics devices are not available for emulation:

EGA64, ATT400, PC3270

The following graphics modes have not been included in the emulation:

CGAC0, CGAC1, CGAC2, CGAC3, MCGGAC0, MCGAC1, MCGAC2, MCGAC3, MCGAMed, MCGAIH, EGALo, EGA64Lo, EGA64Hi, ATT400C0, ATT400C1, ATT400C2, ATT400C3, ATT400Med, ATT400Hi, VGALo, VGAMed, PC3270Hi

New Constants

SVGA, ToshMono

New drivers for SuperVGA and Toshiba plasma screens.

MCGA256

New mode for 320x200 with 256 colors and the driver MCGA.

SVGAComp, TsengLo, TsengMed, TsengHi, TridentLo, TridentMed, TridentHi

Modes for the driver SVGA for 320x200, TSENG 4000 based 640x480, 800x600 and 1024x768 pixels with 256 colors as well as Trident 8900/8900C 640x480, 800x600 and 1024x768 pixels with 256 colors.

TshMonoHi

640x400 mode for the Toshiba driver.
N.B. that the emulation is compatible with version 3.0 of BGI, which has increased the number of available BGI stroke fonts from 4 to 10 compared to version 2.0. Here are all BGI 3.0 fonts:

\[
\begin{align*}
\text{TriplexFont} &= 1; \\
\text{SmallFont} &= 2; \\
\text{SansSerifFont} &= 3; \\
\text{GothicFont} &= 4; \\
\text{ScriptFont} &= 5; \\
\text{SimplexFont} &= 6; \\
\text{TriplexScrFont} &= 7; \\
\text{ComplexFont} &= 8; \\
\text{EuropeanFont} &= 9; \\
\text{BoldFont} &= 10;
\end{align*}
\]

\[
\text{StampPut} = 5;
\]

New drawing mode for overlaying new graphics on existing screen graphics. See "MG - Window mode".

Data Types

Modified Data Types

\[
\text{PointType} = \text{MG}.\text{Point};
\]

New Data Types

\[
\text{Image} = \text{MGImg}.\text{Image}
\]

New data types for images (identical with \texttt{MGImg.Image}) that are used by \texttt{GetImage}, \texttt{PutImage} and \texttt{DisposeImage}. Images of this type also can be directly used as parameters in the \texttt{MGImg} routines.

\[
\text{PaletteArray} = \text{ARRAY}[0..\text{MaxColors}] \text{ OF ShortInt};
\]

Array of palette colors - used in the declaration of the type \texttt{PaletteType}. 

Variables

Variables not Available

\texttt{GraphGetMemPtr}  
\texttt{GraphFreeMemPtr}

Memory management is taken care completely by \textbf{MultiGraphics} with the help of the virtual memory manager in \textbf{VM}. The memory management neither should be, nor can be changed!

New Variables

\begin{verbatim}
revision : Word;
EP : SV.ErrProc;
cap : MG.Capability;
clipImage : BOOLEAN;
\end{verbatim}

\texttt{EP} contains the current error handling procedure of the unit (normally \texttt{SV.GlbEP}).

\texttt{cap} defines the BGI capability of the unit. It is used when you want to give BGI capabilities to virtual screens or devices that have been loaded with \textbf{MultiGraphics} commands.

\texttt{clipImage} specifies if the images are to be clipped correctly (in accordance with \textbf{MultiGraphics}) (\texttt{TRUE}) or in a simplified way as in BGI (\texttt{FALSE}). The default is \texttt{TRUE}. The only reason to change \texttt{clipImage} to \texttt{FALSE} is when the emulation is also to include the \textit{shortcomings} of BGI!
Procedures & Functions

Procedures & Functions not Available

SetVisualPage
SetActivePage

The procedures above are not implemented right now.

ImageSize

The function is used in BGI to decide the size of the memory block that needs to be allocated before calling GetImage. In emulation, the allocation of image memory is taken care of automatically by GetImage, which is why the memory size does not need to be asked for. In contrast to BGI, the image size can also be > 64 KB.

SetGraphBufSize

Not needed in emulation.

Modified Procedures & Functions

GetImage
PutImage

Images must (with emulation) be of the type Image (i.e. of the type MGImg.Image). Examples of reading, drawing and removing images:

Standard BGI:

VAR
  img: Pointer;
  ..
  GetMem(img, ImageSize(100,100,300,200));
  GetImage(100,100,300,200,img);
  ..
  PutImage(0,0,img,CopyPut);
  ..
  FreeMem(img, ImageSize(100,100,300,200));
BGI emulation:

VAR
  img: Image;
  ..
    GetImage(100,100,300,200,img);
  ..
    PutImage(0,0,img,CopyPut);
  ..
    DisposeImage(img);

**InstallUserDriver**

Any specified autodetect function is not used.

**RegisterBGIDriver**

Drivers that are registered must (internally) be of the pointer type **MG.Device**.

**InstallUserFont**

Font files that are to be installed can be of both **MultiGraphics** types and BGI types.

**RegisterBGIFont**

Fonts that are registered must internally be of the type **MG.Font**.

VAR
  fnt : MG.Font;
  fnnum: INTEGER;
  ..
  MG.LoadFont(fnt,'ROMAN.COL',TRUE);
  fnnum:= RegisterBGIFont(fnt);
  SetTextStyle(fnnum,HorizDir,1);
  ..
New Procedures & Functions

**DisposeImage**

```pascal
P DisposeImage(VAR img: Image (*IO*));
```

Deallocates an image attached to the image variable `img`. Compatible with `MGImg.DisposeImage`. 
The unit **MGDebug** makes it easy to supervise and debug heap management och event handling in your **MultiGraphics** applications. You can also use **MGDebug** to log your own data during execution. All information recorded during execution is written to a log file when the application is terminated. This file gets the same name as the application's EXE file, but the file extension is changed to '.LOG'. The log file is created in the same directory as the application EXE file.

For example, if the application is "CREATOR.EXE", then the log file name is "CREATOR.LOG".

You can also write log data whenever you want to a text file of your own choice.

During execution the allocation and deallocation of heap memory is constantly supervised and any error is reported using the normal **SuperVisor** error handling techniques.
Appendix E-2  MGDebug - Debugging

How to Activate MGDebug

You can activate MGDebug either by

1) including MGDebug in the USES list of your application. This will activate the heap supervisor and your own data logging - if used - but not the event logging,

or by

2) compiling the application with the specific debug versions of the units MGErr, MGDebug and possibly MGEv and MGMouse (archived in MGDBG70.EXE). You should put these TPU and TPP files in a separate directory, not the same as the "normal" TPU and TPP files in Multi-Graphics.

Open up the dialog "Options\Directories" in the Pascal IDE and change the "EXE & TPU directory" settings to the debug directory, storing MGErr.TPU, etc., and insert the same directory path first in the "Unit directories" string. Then compile. You can store the debug directory paths in a new option file, for instance "MGDEBUG.TP", to facilitate the switching from non-debugging sessions to debug sessions. Return to the default option files "BP.TP" or "TURBO.TP" when you compile the production version of your application.

Let's suppose that the TPU & TPP files of MultiGraphics are located in the directory "C:\MG\units" and the debug versions in "C:\MG\debug". Then the "EXE & TPU directory" should be changed to read

C:\MG\debug

while "Unit directories" should be set to:

C:\MG\debug;C:\MG\units;C:\bp\units;...etc.

Remember! The debug directory should be placed first in the search paths.

It is not necessary to include the MGDebug unit in the USES clause if you deploy the second method!

The source code of MGDebug is included so you can use the debugger to trace the allocation and deallocation of memory, etc.
SuperVising Allocations & Deallocations

All heap management, i.e. allocation and deallocation, in MultiGraphics is routed to procedures in the unit PasSys. This unit exports the procedure ALLOCATE and the procedure variables IdAllocate and DEALLOCATE:

PROCEDURE ALLOCATE(VAR p : Pointer;
    size: Word):

VAR
    IdAllocate(VAR p : Pointer;
    size: Word;
    id : Word);

DEALLOCATE(VAR p : Pointer;
    size: Word);

The first two parameters of IdAllocate are the same as the parameters to ALLOCATE, and the third parameter is an integer id parameter to identify the specific allocated block or the block type. ALLOCATE, in turn, internally invokes IdAllocate with the id number set to zero. When you compile your application without MGDebug in the USES list or without linking the application with the debug version of TPU or TPP files, the procedure variables are attached to the standard System.GetMem and System.FreeMem. The id is then ignored.

However, if MGDebug is included, or the applications is linked with the debug TPU or TPP files, IdAllocate and DEALLOCATE are attached to modified procedures in MGDebug (DebugIdAllocate and DebugDeallocate) that control and log all allocations and deallocation requests.

As already stated, the conventional memory management in MultiGraphics relies on PasSys.IdAllocate and PasSys.DEALLOCATE. If you wish to supervise your own allocations and deallocations, all calls of New and GetMem should be replaced by calls to PasSys.IdAllocate. All calls of Dispose and FreeMem should be replaced by calls to PasSys.DEALLOCATE. Choose a unique number for each allocated block as an id parameter to IdAllocate or select a unique number to represent the block type. Avoid id number < 1000, which are reserved for
MultiGraphics use.

All heap memory allocations are logged in an internal array, dimensioned to store up to 3000 allocations in real mode and 5460 allocations in protected mode. If this internal memory block array is filled up, you will receive an error message (errHeapBlockArrayFull).

Log data can be written to a file, by simply calling the procedure HeapLog using the file name as a parameter. This happens automatically when the application is terminated.

Let's look at a log file created by a simple application without event handling or any user logged data:

MultiGraphics log file
-----------------------
16-12-1993 12:20
Borland Pascal DOS 16-bit real mode

User log:
--------

Heap log:
---------
Allocated mem now = 156 Peak mem allocated during execution = 68543 bytes
Heap blocks report:
(allocated blocks = !!, deallocated blocks = -- (blockid) = internal block)
Index = 14 ptr = 4936:000C block = 4A95:0008 size = 16352 -- (VM.RamBlock)
Index = 13 ptr = 4936:0010 block = 4937:0008 size = 5600 -- (VM.RamBlock)
Index = 12 ptr = 4925:001D block = 4936:0000 size = 20 -- (VM.BlockStruc)
Index = 11 ptr = 4920:0018 block = 492C:0000 size = 157 -- MG.Window
Index = 10 ptr = 4920:0049 block = 4925:0000 size = 112 -- (MG.ScrPrivate)
Index =  9 ptr = 4916:000E block = 4920:0000 size =  79 -- MG.Screen
Index =  8 ptr = 4916:000A block = 4919:0008 size = 104 !! MG.RGBPalette
Index =  7 ptr = 2FA4:0ACC block = 4916:0000 size =  52 !! MGImg.Image
Index =  6 ptr = 310B:3FDE block = 490E:0000 size = 124 -- MG.Capability
Index =  5 ptr = 2FA4:0B4A block = 4904:0000 size =  157 -- MG.Window
Index =  4 ptr = 48F3:0018 block = 48FA:0000 size = 157 -- MG.Window
Index =  3 ptr = 48F3:0049 block = 48F8:0000 size =  30 -- (MG.ScrPrivate)
Index =  2 ptr = 485C:0320 block = 48F3:0000 size =  79 -- MG.Screen
Index =  1 ptr = 310B:3E73 block = 3F56:0000 size = 39376 -- (VM.RamBlock)
Index =  0 ptr = 2FA4:0B18 block = 3DD6:0000 size =  6144 -- (MG.Buffer)

Events log
----------
Nothing to report!

In this example you will find the amount of heap memory that is still in use when the application terminates (here 156 bytes), the peak number of bytes
that has been allocated at any time (here 68543 bytes), as well as a line for each allocated block, sorted in reverse order so that the last allocated block is listed first.

Each block is specified by it's ordinal number ("Index = "), the address of the pointer that was used when the block was allocated ("ptr = "), the address of the allocated memory block ("block = "), the size of the allocated block ("size = "), the code "--" if the block has been deallocated and "!!" if the block is still allocated and finally the id number of the block or block type. All data types in MultiGraphics have been replaced by names. Internal data types are enclosed inside parenthesis. In our example we have apparently forgotten to deallocate an image using MGImg.DisposeImage.

During execution the following errors are reported:

- attempts to deallocate memory blocks with a size parameter different from the allocated block size (errInvalidBlockSize).
- attempts to deallocate blocks that have already been deallocated (errAlreadyDeallocated).
- attempts to deallocate blocks that have never been allocated (errNotAllocated).

Additionally, MGDebug can also raise warnings whenever the following, potential errors appears:

- the pointer variable used at allocation is already in use and attached to another memory block (warnPointerInUse).
- the pointer variable used at deallocation is not the same as the pointer used at allocation time (warnNotOriginalPointer).
- the pointer used at deallocation, resides in the memory area of the memory block to be deallocated. (warnPointerInBlock).

To activate the warnings you have to set the exported warn variable to TRUE.

MGDebug.warn := TRUE;

If you want to supervise the memory management in greater detail, you can compile MGDebug with debug information.
Supervising Events

If you link your event based application with the MultiGraphics debug versions, all events that occur will be logged in an internal array. When you call the procedure EventsLog all logged data is written to a text file of your own choice. At the termination of the program all data is automatically written to the default log file. You can log up to 3600 events. If the log array is filled up, new events will overwrite old one's from the start of the array. You can clear the log data by invoking ResetEventsLog.

Events log
----------

Timer ticks at start........................... 825827
Total events to interrupt queue............... 1268
Total events stored in interrupt queue....... 1268
Total events to event queue................... 1267
Total events stored in event queue.......... 1267
Peak events in interrupt queue............... 2 of 8
Peak events in event queue.................... 52 of 256
Clock generator: Total events................. 800
Mouse generator: Total events............... 468
Current pointer redraw time (1000=scan time)... 400
Total events processed by mouse event handler 467
Total updates of mouse windows position...... 304
Total mouse pointer trace calls............... 288
Total automatical pointer protections......... 34
Total calls to HidePointer.................... 28
Total times of redrawing mouse pointer....... 328

All events logged in the event queue:
-------------------------------------

*******************************************************************
- = events overwritten (queue filled up)

Clock events:
-------------

CT = Clock Tick OA = Onetime Alarm RA = Repeated Alarm

Keyboard events:
----------------

KP = Key Pressed

Shift keys:
-----------

RS = Right Shift LS = Left Shift CC = Ctrl AA = Alt
LC = Left Ctrl LA = Left Alt SR = Sys Rq
Toggle keys:
-------------
SL = Scroll Lock NL = Num Lock CL = Caps Lock

Mouse events:
-------------
MM = Mouse Moved LP = Left button Pressed
LR = Left button Released RP = Right button Pressed
RR = Right button Released MP = Middle button Pressed
MR = Middle button Released LD = Left button Double click
RD = Right button Double click MD = Middle button Double click

Mouse buttons:
-------------
LB = Left Button RB = Right Button MB = Middle Button pressed

# ticks - events info

<table>
<thead>
<tr>
<th>#</th>
<th>Event Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLOCK :CT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>CLOCK :CT</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>885</td>
<td>CLOCK :CT RA repeated: 2 3</td>
<td></td>
</tr>
<tr>
<td>886</td>
<td>MOUSE :MM buttons: RB shifts: x,y:314,111</td>
<td></td>
</tr>
<tr>
<td>887</td>
<td>CLOCK :CT RA repeated: 2 3</td>
<td></td>
</tr>
<tr>
<td>888</td>
<td>MOUSE :MM buttons: RB shifts: x,y:314,110</td>
<td></td>
</tr>
</tbody>
</table>

First MGDebug reports some total statistics, including "Peak events ..", which are particularly important to observe. If the first value is equal to, or greater than the second value, then the event queue is filled up. This can happen if the applications ignores to read data from the event queue or if it reads the event queue too slow. In our example everything is OK!

Below this header, you will find a report on all events, with their ordinal numbers, counted from the first logged event, the clock time when the events occured, the event generators and finally event specific information.
If the event queue is filled up, events that are overwritten is marked with a ",-". Those events are lost.

User Log

MGDebug declares two arrays, `logStr` with up to 100 text strings and `log` with up to 100 integers. These arrays can be used to log your own data during execution. User data can be written to a text file by calling `UserLog`. If there are any data in the arrays, these will be automatically reported to the default log file at program termination.
Compatibility Questions

APPENDIX F

Compatibility Questions

PCX Images

MultiGraphics can read and create images using the following PCX formats:

Monochrome images:
One bit per pixel, one color plane.
Monochrome images can be used in all graphics modes.

Images with 2 - 16 Colors
2 - 16 color images with or without palette information, one bit per pixel and one to four color planes.
Can be used with EGA, VGA, MCGA, SuperVGA and 8514/A modes as well as with matrix printers, HP DeskJet and HP PaintJet color modes.
If the palette information is missing in the image file, the default palette of DOS and MultiGraphics is used.
Images with 256 Colors

256 color images with our without palette information, eight bits per pixel and one color plane.

Can be used with MCGA, SuperVGA and 8514/A modes.

PCX Images Created by Commercial Drawing Programs

Paintbrush, Paintbrush IV, Publishers Painbrush

Those are the DOS versions of Paintbrush and can be used to create images that are fully compatible with MultiGraphics. Normally the default palette of DOS & MultiGraphics is used.

Corel Draw 4.0 for Windows

Monochrome PCX Images

Can be used with MultiGraphics.

16 Color PCX Images with Palette

Four bits per pixel, one color plane. These images cannot be read by MultiGraphics directly, but you can convert them to the correct format by loading the images from Windows Paintbrush or DOS Paintbrush and then save them again. The image is then stored with one bit per pixel, four color planes and palette information. The image palette is different from the DOS & MultiGraphics default so you have to set the application palette to the image palette or the Windows default palette.
256 Color PCX Images with Palette

Can be used with **MultiGraphics** 256 color modes. You have to set the palette.

**MicroGrafx Designer 4.0 for Windows**

Monochrome PCX Images

Can be used with **MultiGraphics**.

256 Color Images with Palette

Can be used with **MultiGraphics** 256 color modes. You have to set the palette.

**Windows Paintbrush**

Monochrome PCX Images

Can be used with **MultiGraphics**.

16 Color Images with or without Palette

You can only create this type if Window is running in a 16 color mode (EGA or VGA). PCX images are compatible with **MultiGraphics**, but the palette is different.

256 Color Image with or without Palette

Can be used with **MultiGraphics** 256 color modes. You have to set the palette.
Here are the default palettes for DOS/MultiGraphics, Microsoft Windows 3.x and Windows Paintbrush (read the comments below):

<table>
<thead>
<tr>
<th>Index</th>
<th>DOS/MG</th>
<th>Windows</th>
<th>Win Paintbrush</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>black</td>
<td>black</td>
<td>black</td>
</tr>
<tr>
<td>1</td>
<td>blue</td>
<td>red</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>green</td>
<td>green</td>
<td>green</td>
</tr>
<tr>
<td>3</td>
<td>cyan</td>
<td>yellow</td>
<td>cyan</td>
</tr>
<tr>
<td>4</td>
<td>red</td>
<td>blue</td>
<td>red</td>
</tr>
<tr>
<td>5</td>
<td>magenta</td>
<td>magenta</td>
<td>magenta</td>
</tr>
<tr>
<td>6</td>
<td>brown</td>
<td>cyan</td>
<td>yellow</td>
</tr>
<tr>
<td>7</td>
<td>light gray</td>
<td>dark gray</td>
<td>dark gray</td>
</tr>
<tr>
<td>8</td>
<td>dark gray</td>
<td>light gray</td>
<td>light gray</td>
</tr>
<tr>
<td>9</td>
<td>light blue</td>
<td>light red</td>
<td>light blue</td>
</tr>
<tr>
<td>10</td>
<td>light green</td>
<td>light green</td>
<td>light green</td>
</tr>
<tr>
<td>11</td>
<td>light cyan</td>
<td>light yellow</td>
<td>light cyan</td>
</tr>
<tr>
<td>12</td>
<td>light red</td>
<td>light blue</td>
<td>light red</td>
</tr>
<tr>
<td>13</td>
<td>light magenta</td>
<td>light magenta</td>
<td>light magenta</td>
</tr>
<tr>
<td>14</td>
<td>light yellow</td>
<td>light cyan</td>
<td>light yellow</td>
</tr>
<tr>
<td>15</td>
<td>white</td>
<td>white</td>
<td>white</td>
</tr>
</tbody>
</table>

The precise RGB values differ slightly between these palettes.

**Windows Paintbrush** has some oddities; when you create a new image from scratch and then saves this image as a PCX file, Paintbrush will use the palette in the rightmost column above. However, the created PCX file lacks palette information, which implies that you have to set the palette for yourself. A better solution, and here comes the bizarre, is to load the saved image once again (still running Windows Paintbrush) and then save the image another time, and *voila* the new image is stored with the standard Window palette found in the middle column and with palette information included (we think this is a Paintbrush bug and not a feature).

You can set the default Windows palette by invoking

```pascal
MGCol1.SetMSWindowsPalette;
```

If the image contains palette information, you can also use

```pascal
MG.SetRGBPalette(MGImg.ImageRGBPalette(img));
```
The VESA standard, or more correctly the "Super VGA BIOS Extension" adapted by VESA ("Video Electronics Standards Association") in 1989 (1.0) and renewed in 1990 (1.1) and 1992 (1.2), makes it possible to utilize the full capacity of a number of SuperVGA adapters without any hardware restrictions. VESA presents a uniform programming API for querying the capacity of the adapter, as resolution, number of colors, etc., and to activate a desired graphics mode. The VESA standard has been implemented as an extension of the standard VGA BIOS.

New graphics adapters (S3, Cirrus, etc.) usually have the VESA support in their BIOS and the VESA modes can be used without any preparations. Older SuperVGA adapters often lack direct VESA support and to use VESA with these cards you have to first load a suitable VESA driver (a COM or EXE file), for example in AUTOEXEC.BAT.

You will find a large number of hardware specific VESA drivers in the VESA directory on one of the MultiGraphics disks. Moreover there is also a "universal" driver, UNIVESA.EXE, which will automatically detect a large number of adapters and install the suitable VESA support. However, UNIVESA is written with 386 code, and thus can't be used with older processors. UNIVESA also requires slightly more memory (9 KB) than the hardware tailored drivers.

The test program GAModes presents a concentrated information on the supported graphics modes. With VESAINFO.EXE you can create a detailed report on the VESA support.
Appendix H-2

**VESAINFO <Enter>**

displays all available VESA modes to the screen, while

**VESAINFO filename <Enter>**

generates a report to a text file.

The drivers **VGA.MGV/VGA.XGV** and **VGA.MGA/VGA.XGA** include support for 16 colors VESA mode 258, resolution 800x600 pixels. See "MGVGA - VGA" for more information.

The drivers **SVGA.MGV/SVGA.XGV, SVGA.MGA/SVGA.XGA** and **SVGA.MGH/SVGA.XGH** include support for all 256 colors VESA modes supported by the hardware. See "MGSVGA - SuperVGA" for more information.

The VESA standard makes it feasible to execute **MultiGraphics** applications on most PC's, without limiting the support to the least common denominator (usually VGA 640x480, 16 colors). A suitable mode can be detected with the procedures **MG.GetFirstDeviceModeInfo** and **MG.GetNextDeviceModeInfo** - see "MG - Devices" for more information. **GAModes** shows the technique to use.
Animation on the PC

This speech was held at the Borland International Developers Conference in beautiful Monterey, California, April 1992. The speech was accompanied by a computer presentation demonstrating many techniques for doing graphics animations.

The computer show is found on the disk labelled "Animation on the PC" in the self-extracting file ANIMATE.EXE. First create an empty directory on your hard disk, then put the animation disk in, for example, drive A: and unpack the files like this:

A:ANIMATE <Enter>

The presentation requires the following system support:

- At least 4 MB free space on the hard disk
- A VGA compatible graphics adapter
- A mouse with a mouse driver loaded
- A minimum of 800 KB free expanded EMS memory (use EMM386 or another memory manager to create this at start-up of computer).

Start the presentation with

ANIMATE <Enter>

Please, wait for a while until the presentation is up and running. Use <Enter> to step forward in the presentation. Use <Esc> to quit the presentation.

In the upper, right hand corner, you will see two numbers, separated by a colon ":". The first number is the sequential number (1,2,3,...) of the current presentation page, while the last number is an internal number in the presentation command file (see EANIMATE.TXT). If you read the speech, that follows, in parallel with the computer presentation you will find the those numbers in the margins of the text.

When you are at the bottom of a presentation page, you can go to the next page by hitting <Enter> or jump to a specific page by writing it's presentation page number and then <Enter>.

The presentation runs a number of example programs (compiled to EXE files). You will find the names of these programs and where and when they are executed, in the margins to the speech.

The most common commands in the demo programs are:
<Enter> Step forward in the presentation
<Space> Toggle between slow motion and full speed
          (please note that it may take a short time
          before the toggle is activated. Just press
          once)
<+>/<-> Faster/slower animation  (does not work
          in slow-motion position)
<Esc> Quit demo and return to presentation
      program.

You will find the details on all demo programs at the end of this text (after the speech
      text).

The demo programs Pascal sources are included on the "Animation" disk. The
      programs will compile with Turbo Pascal or Borland Pascal.
Animation on the PC

by Lars Gustafsson, DATABITEN AB, Sweden

In this paper I will describe some basic techniques for animated computer graphics, techniques that you can use on an every-day PC displaying VGA graphic. The methods are not tied to a specific programming language, but I will use Turbo Pascal for DOS in this presentation.

What is animation?

The word animation means to bring to life, to include motion and change in the presentation. To move from static displays to computer animated displays, is historically analog to the step from static pictures to moving pictures made in the late nineteenth century.

Our ambition is to imitate or emulate reality as closely as possible, or at least to fool the user to believe he or she is watching a TV or a movie.

Different aspects of animation

We wish, depending on the situation, to animate the full screen or just a smaller part of it. At other times we wish to move fixed objects in front of a static background or changing objects in front of a static background. If you look at movie cartoons, the latter approach is the most common because it simplifies and reduce the labor.

In the concept of animation we include all kinds of changes, not just position changes, but also change of size, shape, color, transparency, etc.
Computer animation comes close to multi-media, but the difference is that computer animation is fully computer generated, not just a playback of a recording using special hardware.

In this paper we will concentrate on basic drawing techniques of computer animation. Other elements as timing and the details of managing animated motions are equally important, but will be mentioned only briefly.

Perception comes to help

Reality (at least at our macroscopic level) is continuous, not discrete, i.e., all motions, colors and changes are continuous, without leaps. However in our computer generated world both memory (video memory) and time are discrete entities, and therefore our first conclusion is that real computer animation is impossible!

To our help comes the limitations of the human perception and the ability of the brain to interpolate and fill-in between otherwise discrete events. That is why we believe we are looking at a full-color display, when we in reality only looks at different mixes of just three colors (red, green and blue). This also explains why we think we are looking at real motions on the TV, when we in reality look at a rapid sequence of static pictures frames.

TV and movie techniques show that we need to display fifty or more pictures each second to fool our brain to believe it is looking at real motion, and to avoid irritating jerkiness and flickering. To reduce the number of pictures TV and movies display the same picture twice, which reduce the number of different pictures each second to 25 - 30.

Another important aspect of animation is the capability of the brain to interpret a flat 2-dimensional display as a 3-dimensional world. Using shadows enhance the 3-D effect.

The display hardware

Before digging into different techniques of animation we will stop for a while at the display technology. On the PC we are storing our pictures as digital information in a video memory or frame buffer. Colors are represented by pure numbers, and if we're looking at the basic level, of just zeroes and one's. This is called bitmapped graphics or raster graphics.

Using special hardware on the display adapter (CGA, EGA, VGA, Hercules, etc.) this video memory is read or scanned fifty times or more every second, and the content is translated and sent to the screen. This process is called scan conversion and it take place all the time without your intervention. The screen display is thus refreshed at least fifty times per second.
The electron beam draws the image starting from the uppermost left corner and in rows down to the rightmost lower corner. At this position the beam is turned off and repositioned to the uppermost left corner. There is a short delay between turning the beam off and the drawing of the next frame. This delay is called the (vertical) retrace interval and amounts to approximately 1/1000 of a second.

The diagram illustrates the display scanning and retrace.

If possible, it is desirable to restrict certain kinds of drawing, for example modifications of the video memory, to this retrace interval. Then we can hide certain kinds of undesirable changes from being displayed on the screen.

There are two different methods for synchronization with the start of the retrace interval. The first one is a polling or asking one, i.e., we're waiting for retrace to occur. Here is some Pascal code useful on the EGA, VGA and SuperVGA:

```
PROCEDURE WaitForRetrace;
VAR
  adressPort: Word ABSOLUTE $0040:$0063;
  statusPort: Word;
BEGIN
  statusPort:= adressPort + 6;
  REPEAT UNTIL 8 AND port[statusPort] = 0;
  REPEAT UNTIL 8 AND port[statusPort] <> 0;
END;
```

By calling this procedure before making sensible drawing we can synchronize our drawing with the start of the retrace time.

Of course we're wasting time, just sitting there waiting for retrace to start. An even better approach is to make the video adapter generate a hardware interrupt at the start of the retrace interval. Both EGA and VGA are supporting this. We take care of the interrupt with an interrupt service routine. Between interrupts we can do useful things as preparing the next frame, make computations and disk accesses etc..

On some video adapters it is possible to ask if the scan conversion has passed a certain pixel or row. With this information we can start to modify the video memory when this pixel or this row has been scanned. We can trail behind the beam and get more time to modify the video memory.

The retrace interval is very short, and often too short even when we use fast 386s or
486s. Perhaps you may exploit the fact that, if your drawing is synchronized with retrace, there is more time to update the lower part of the screen than the upper part!
Full screen animation

The most direct and brute-force approach is simply to imitate the movie technique and thus generate 20 - 30 new screen frames each second. Using this technique we will be able to support all kinds of animations. We can animate digitized and pre-stored bitmapped pictures or we can redraw the full screen for each frame.

Here is the simple animation algorithm:

1. Draw next screen frame
2. Delay
3. Repeat from step 1.

If the frames are at hands as ready-made bitmaps, here is the simple Pascal code (first using MultiGraphics and then BGI graphics) to achieve this full screen animation:

**MultiGraphics:**

```pascal
VAR
  .frame: ARRAY[1..n] OF MGImg.Image;
  ...
MG.SetDrawMode(MG.replace); MG.SetPos(0,0); FOR i:= 1 TO n DO MGImg.DrawImage(frame[i]);
```

The `replace` drawing mode effectively make each new image replace the preceding one.

**BGI:**

```pascal
TYPE
  Buffer = ARRAY[0..max] OF BYTE;
VAR
  frame: ARRAY[1..n] OF Buffer;
  ...
FOR i:= 1 TO n DO PutImage(0,0,frame[i],copyPut);
```

The `copyPut` drawing mode effectively make each new image replace the preceding one.

However there are at least two con's of this simple approach:

- Speed
- Memory

We need an extremely fast computer, especially if we redraw the screen for each frame. We also need a lot of memory if the frames are stored as ready-made bitmaps.

Both problems increase with the power of two when increasing image size and screen resolution and increase directly with the increase of color capacity. Just compare the total information contained in a 256 colors 1024x768 SuperVGA image to a monochrome 320x200 image. Yes, the first image has more than 100 times more information!
Now let's turn to more restricted animations, animating a smaller object on an otherwise static screen. This is not a totally unrealistic approach - looking around you will find static buildings, furniture, mountains, etc. and few living objects.

**Static rectangular objects**

Our first assumption is that the animated object (or objects) is rectangular and does not move across the screen. The object is stored as a frame sequence of bitmap images, or we may draw these frames at high speed during run-time.

With these restrictions in mind we can use the same animation technique as described above for full screen images, i.e.:

1. **Draw next image frame**
2. **Delay**
3. **Repeat from step 1.**

**MultiGraphics:**

```
VAR
  frame: ARRAY[1..n] OF MGImg.Image;
  ...
MG.SetDrawMode(MG.replace); MG.SetPos(x,y);
FOR i:= 1 TO n DO BEGIN
  MGImg.DrawImage(frame[i]);
  Delay(dt);
END;
```

**BGI:**

```
VAR
  frame: ARRAY[1..n] OF Buffer;
  ...
FOR i:= 1 TO n DO BEGIN
  PutImage(x,y,frame[i],copyPut);
  Delay(dt);
END;
```

If the object is not too large it is recommended to synchronize drawing with the vertical retrace to reduce flicker, so:

**MultiGraphics:**

```
FOR i:= 1 TO n DO BEGIN
  WaitForRetrace; or standard MG.WaitNoFlicker;
  MGImg.DrawImage(frame[i]);
  Delay(dt);
END;
```

**BGI:**

```
FOR i:= 1 TO n DO BEGIN
  WaitForRetrace;
  PutImage(x,y,frame[i],copyPut);
  Delay(dt);
```
Under a few conditions, as reasonable object size and number of colors, this technique is highly useful.

Moving object, unprotected background

Now we will relax our restrictions on object motion. In general this pose a new problem of protecting the background when the object moves in front of it. The problem arises because both background and foreground information is stored in the same two-dimensionally organized video memory.

To simplify, we will first assume that the background does not need to be protected, maybe it's black. However we are not freed from the obligation to clean up after each frame has been displayed and the object moves to a new position. Otherwise the screen will be cluttered with object fossils. Here is our algorithm:
Draw - erase

1. Draw next image frame
2. Delay
3. Erase image
4. Set next position
5. Repeat from step 1

MultiGraphics:

```pascal
MG.SetDrawMode(MG.replace);
MG.SetPattern(backPattern);
FOR i:= 1 TO n DO BEGIN
  MG.SetPos(x,y);
  MGImg.DrawImage(frame[i]);
  Delay(dt);
  {w,h = image width & height}
  {WaitForRetrace;}
  MG.FillRect(x,y,w,h);
  getNextPos(x,y); {user defined getNextPos(VAR x,y:INTEGER)}
END;
```

BGI:

```pascal
FOR i:= 1 TO n DO BEGIN
  PutImage(x,y,frame[i],copyPut);
  Delay(dt);
  {WaitForRetrace;}
  SetFillStyle(backPattern,backColor);
  {w,h = image width & height}
  Bar(x,y,x+w-1,y+h-1);
  getNextPos(x,y); {user defined getNextPos(VAR x,y:INTEGER)}
END;
```

We first draw the image at one position, than take a short break, erase our image and move to a new position, etc.. Of course it is extremely important that the operations of erasing and redrawing are accomplished in a short time, because between erase and redraw there is no object on the screen! Otherwise we will see an annoying flickering and the object will be displayed with an averaged and reduced intensity. If possible, the best is to accomplish the erase and redraw operations during retrace.

![Draw - erase animation](image_url)
Draw & erase

To avoid the erase operations we may occasionally exploit a little trick. Let the objects erase themselves! We accomplish this by increasing the size of the object images, so that the each image frame has a surrounding border of black or background color. If the object movements between displayed frames are less than the width of its border, each new frame will effectively erase the previous object image. Here is the simplified Pascal code:

**MultiGraphics:**

```pascal
FOR i := 1 TO n DO BEGIN
    MG.SetPos(x,y);
    {WaitForRetrace;}
    MGImg.DrawImage(frame[i]);
    Delay(dt);
    GetNextPos(x,y);
END;
```

**BGI:**

```pascal
FOR i := 1 TO n DO BEGIN
    {WaitForRetrace;}
    PutImage(x,y,frame[i],copyPut);
    Delay(dt);
    GetNextPos(x,y);
END;
```

This technique is useful but has its drawbacks. Don't fool yourself with underestimating the real size of the border area! Under the assumption that the drawing time is proportional to the image size or area, we make a small analysis of when the image border becomes as large as the object image. If the border area is larger than the object area, and if the erase operation is as fast as the drawing operation, it will be faster to use "draw - erase" than "draw & erase."

With a square shaped object of width and height $s$, and a border of width $d$, the following condition must hold:

$$(s + 2d)^2 - s^2 < s^2$$

that is simplified to

$$d < 0.2s$$

So if we need a surrounding border to each side of the object (this is not always necessary if the motion is restricted) the "draw & erase" procedure is only useful if each frame repositioning is relatively small, less than 1/5 of the object size.
Moving object, protected background

The situation becomes more complicated when we have to face a real background that has to be protected. Neither the "draw - erase," nor the "draw & erase" schemes now meet our demands.

Well, it is still possible to use "draw & erase," if the background has a repeating pattern, and this pattern is also stored in the surrounding object border, and finally all object displacements are in multiples of the pattern cycle interval.

The general procedure is first to save the altered background in off-screen memory, then draw our object frame, and finally restore the saved background.

Save - draw - restore

1. Save the frame background
2. Draw next frame image
3. Delay
4. Restore the saved background
5. Set next position
6. Repeat from step 1.

The corresponding Pascal code looks like this:

**MultiGraphics:**

```pascal
VAR
  background: MGImg.Image;
  frame      : ARRAY[1..n] OF MGImg.Image;
  ..
FOR i:= 1 TO n DO BEGIN
  MG.SetPos(x,y);
  MGImg.ReadImage(background,w,h);
  MGImg.DrawImage(image[i]);
  Delay(dt);
  {WaitForRetrace;}
  MGImg.DrawImage(background);
  MGImg.DisposeImage(background);
  GetNextPos(x,y);
END;
```

**BGI:**

```pascal
VAR
  background: Buffer;
  frame      : ARRAY[1..n] OF Buffer;
  ..
FOR i:= 1 TO n DO BEGIN
  GetImage(x,y,x+w-1,y+h-1,background);
  PutImage(x,y,image[i],copyPut);
  Delay(dt);
  {WaitForRetrace;}
  PutImage(x,y,background,copyPut);
  GetNextPos(x,y);
END;
```
This technique is suited for animation of all kinds of rectangular animated objects, both bitmapped and real-time drawing. The scheme totals to three copy operation for each object frame and naturally it is critical that these operations (step 1, 2 and 4) are accomplished very fast, if possible during retrace. If this fails the user will perceive flicker and a feeling of a faded and even transparent object.

There remains one big problem, namely what to do if our objects are not rectangular shaped? Real world objects may have any shape and may even have holes.

There is no problem if we draw the objects ourselves directly on screen, but if we copy the frame images from off-screen bitmaps, we will affect everything in a rectangular area, as there is no other effective way to store and copy bitmaps.

To find a solution to this problem we have to introduce some ideas from mathematics.
Logical algebra

In 1854, the great English mathematician, George Boole, invented a new logical algebra, mainly to do calculations on logical expressions. In honor of Boole we also call this discipline for Boolean algebra.

In short the Boolean algebra guide us to evaluate and simplify expressions containing logical operators and operands. Logical expressions are two-valued, i.e., they can only result in the values true or false.

The computer scientists in the forties and fifties almost immediately recognized that this simple logic perfectly matched the digital, binary computer design. The Boolean expressions find their use in both logical program constructions (IF statement, etc.) and in computations on data (for example image data), which are ultimately stored as ones and zeroes, the "quarks" of computation.

In computer logic we usually think of the digit "0" as the truth value "false," and "1" as "true." Truth values may be combined to expressions using logical or boolean operators, the most commonly used are "NOT," "AND" and "OR" - all well-known to every programmer. Their function is easily described by their truth tables:

**NOT**

<table>
<thead>
<tr>
<th>Logical</th>
<th>Bitwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT false -&gt; true</td>
<td>NOT 0 -&gt; 1</td>
</tr>
<tr>
<td>NOT true -&gt; false</td>
<td>NOT 1 -&gt; 0</td>
</tr>
</tbody>
</table>

**AND**

<table>
<thead>
<tr>
<th>Logical</th>
<th>Bitwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>false AND false -&gt; false</td>
<td>0 AND 0 -&gt; 0</td>
</tr>
<tr>
<td>false AND true -&gt; false</td>
<td>0 AND 1 -&gt; 0</td>
</tr>
<tr>
<td>true AND false -&gt; false</td>
<td>1 AND 0 -&gt; 0</td>
</tr>
<tr>
<td>true AND true -&gt; true</td>
<td>1 AND 1 -&gt; 1</td>
</tr>
</tbody>
</table>
OR

<table>
<thead>
<tr>
<th>Logical</th>
<th>Bitwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>false OR false</td>
<td>0 OR 0</td>
</tr>
<tr>
<td>false OR true</td>
<td>0 OR 1</td>
</tr>
<tr>
<td>true OR false</td>
<td>1 OR 0</td>
</tr>
<tr>
<td>true OR true</td>
<td>1 OR 1</td>
</tr>
</tbody>
</table>

There is another operator, which in logic is not used very often, but which we will find a use for right now. This operator is called "exclusive or" shortened to "XOR." Here is the corresponding truth table:

XOR

<table>
<thead>
<tr>
<th>Logical</th>
<th>Bitwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>false XOR false</td>
<td>0 XOR 0</td>
</tr>
<tr>
<td>false XOR true</td>
<td>0 XOR 1</td>
</tr>
<tr>
<td>true XOR false</td>
<td>1 XOR 0</td>
</tr>
<tr>
<td>true XOR true</td>
<td>1 XOR 1</td>
</tr>
</tbody>
</table>

The only difference between OR and XOR is on the last line in their truth tables. 1 OR 1 -> 1, but 1 XOR 1 -> 0. This explains the name "exclusive or" and this difference makes XOR very useful in computer animation.

XOR animation

We may think of every video pixel (picture element) stored in video-memory or in an off-screen image, as having only two values, namely "on" (1) or "off" (0). "1" stands for the foreground color (for example white) and "0" for the background color (for example black). With this interpretation in mind, let’s look on the truth tables again.

We interpret the first operand in the first column as the pixel value of the image frame, and the second operand as the pixel values in screen video memory (the background), before the operation. The expression value is the result in video memory of combining the two pixels according to the specific logical rule. Here is the result for

XOR:

"black image" XOR "black screen" gives "black screen"
"black image" XOR "white screen" gives "white screen"
"white image" XOR "black screen" gives "white screen"
"white image" XOR "white screen" gives "black screen"

The screen output may seem remote from the original image, but you should make these observations:
On a black screen background the image is visible with its original colors. On a white screen background the colors of the image are *inversed*, but the object is still visible. With full color images the color effects will be more complex, but the image is visible, regardless of the background.

Still, what is the point in this, if our goal is animation?

Now, XOR has an unexpected behavior of nullifying itself, when applied a second time! Let’s see if we draw the same object frame twice (without moving the object), using XOR:

First drawing

<table>
<thead>
<tr>
<th>Object image</th>
<th>Screen background</th>
<th>New screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 XOR</td>
<td>0</td>
<td>-&gt; 0</td>
</tr>
<tr>
<td>0 XOR</td>
<td>1</td>
<td>-&gt; 1</td>
</tr>
<tr>
<td>1 XOR</td>
<td>0</td>
<td>-&gt; 1</td>
</tr>
<tr>
<td>1 XOR</td>
<td>1</td>
<td>-&gt; 0</td>
</tr>
</tbody>
</table>

Note that the screen background is fully restored to its original look after the second drawing. This is like mirrors; one mirror swaps right and left, two mirrors make everything "right" again.

The XOR operator has the valuable quality of cleaning up after itself. Here is our simple animation algorithm:

**XOR-draw**

1. XOR-draw next image frame  
2. Delay  
3. XOR-draw the same image frame again  
4. Set next position  
5. Repeat from step 1.

**MultiGraphics:**

```
MG.SetDrawMode(MG.complement);
FOR i:= 1 TO n DO BEGIN
  MG.SetPos(x,y);  
  MGImg.DrawImage(frame[i]);
```

DATABITEN
Delay(dt);
MGImg.DrawImage(frame[i]);
GetNextPos(x,y);
END;

Note the new drawing mode called complement in MultiGraphics.

**BGI:**

FOR i:= 1 TO n DO BEGIN
  PutImage(x,y,frame[i],xorPut);
  Delay(dt);
  PutImage(x,y,frame[i],xorPut);
  GetNextPos(x,y);
END;

Note the new copy mode called xorPut in BGI graphics.

The "XOR-draw" even works with color graphics, as this is ultimately represented by zeroes and ones. The color effects of xor'ing are however more difficult to predict.

Another strong point in favor of "XOR-draw" is that it works equally well with application drawn objects, and that the objects do not need to be rectangular, instead they may have any shape. The XOR-technique is very popular in animations, in windowing systems (moving the windows), drawing or painting programs (drawing, resizing and moving objects) and for cursors. One very popular use is called "rubber-banding."

According to an article in Dr. Dobbs Journal someone (no names) has even tried to patent the "XOR cursor" technique so maybe I'm revealing trade secrets. Well, I think we should honor Mr. Boole instead who is the real inventor.

To conclude these are the strong points of the "XOR-draw" technique:

+ Simple
+ Fast
+ Any object shapes
+ Both bitmap copying and application drawn graphics

The only weak point is this

- Object colors depend on background
The latter problem is especially important if the background has no large homogeneous areas. Then the xor'ed objects tend to look blurry.

Masking technique for raster graphics

We turn back to the "save-draw-restore" scheme, which compared to "XOR-draw" has the important property to preserve object colors, that is the objects look like they should. However we have to find some method to cope with irregular, non-rectangular shapes. We have to specify in some way, which parts of a rectangular bitmap that belongs to the object and should be copied, or differently expressed, which parts of the screen that should be protected.

Once again we turn back to Mr. Boole and his logical algebra. To our array of image frames we add another array of *monochrome* bitmap *masks*, one for each different frame. These masks are filled (corresponding bits set to 1) where we want to protect the background, and empty (corresponding bits set to 0) where we want to draw the objects.

The first step is to draw the mask frame, using the logical **AND** mode. Here is the truth table of the **AND** operator again:

<table>
<thead>
<tr>
<th>mask</th>
<th>Screen background</th>
<th>New screen</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AND 0</td>
<td>0 -&gt; 0</td>
<td>background erased</td>
</tr>
<tr>
<td>0</td>
<td>AND 1</td>
<td>0 -&gt; 0</td>
<td>background erased</td>
</tr>
<tr>
<td>1</td>
<td>AND 0</td>
<td>0 -&gt; 0</td>
<td>background protected</td>
</tr>
<tr>
<td>1</td>
<td>AND 1</td>
<td>0 -&gt; 1</td>
<td>background protected</td>
</tr>
</tbody>
</table>

As you see, the background of the screen is not modified where the mask contains "1," but is erased where the mask contains "0." The visible result will be a "hole" (or several holes) in the screen background. This hole is there to receive the content of our object.

The next step is to draw the object frame, but only in the hole or holes on the screen. Let's examine the truth table of the **OR** operator. It looks like this when writing to the holes:

<table>
<thead>
<tr>
<th>image</th>
<th>Screen holes</th>
<th>New screen</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OR 0</td>
<td>0 -&gt; 0</td>
<td>image fills hole</td>
</tr>
<tr>
<td>1</td>
<td>OR 0</td>
<td>0 -&gt; 1</td>
<td>image fills hole</td>
</tr>
</tbody>
</table>

The holes are filled with the image colors. That's exactly what we want. What then about the screen background outside the holes, i.e., the parts protected by the mask? Notice that the image is always black here:
The background is not modified. Perfect! Here is our full algorithm:

Save - AND-mask - OR-draw - restore

1. Save the frame background
2. AND-draw next frame mask
3. OR-draw next frame image
4. Delay
5. Restore the saved background
6. Set next position
7. Repeat from step 1.

MultiGraphics:

FOR i:= 1 TO n DO BEGIN
    GetImage(x,y,x+w-1,y+h-1,background);
    PutImage(x,y,mask[i],andPut);
    PutImage(x,y,frame[i],orPut);
    Delay(dt);
    {WaitForRetrace;}
    PutImage(x,y,background,copyPut);
    GetNewPos(x,y);
END;

BGI:

FOR i:= 1 TO n DO BEGIN
    GetImage(x,y,x+w-1,y+h-1,background);
    PutImage(x,y,mask[i],andPut);
    PutImage(x,y,frame[i],orPut);
    Delay(dt);
    {WaitForRetrace;}
    PutImage(x,y,background,copyPut);
    GetNewPos(x,y);
END;

Note the use of two new copy modes, namely andPut and orPut.

It is imperative that steps 1, 2, 3 and 5 are accomplished quickly, as usual during retrace. Otherwise we get the same undesirable phenomena of flicker, reduced intensity and transparency.
The combination of AND masking and OR drawing give us the tools to draw any object shapes, even those with holes. This mask technique is the most general of all described methods for bitmapped raster animations. It is widely used in such an everyday animation as the drawing of mouse cursors.

Controlling transparency

Black parts of the image frame (bits = 0) are painted as solid black if the corresponding bits in the mask are set to zero, but are transparent where the mask bits are set to one.

However the mask does not necessarily need to match the image. Here is an example:

We want a white mouse cursor, but want the cursor to be visible regardless of the screen background color. Now, how can a white cursor be visible if the background is white?

The solution is simple; just make the mask hole a little bit larger than the image outline. This produce a black contour surrounding the object, visible whatever the background. You may also explore the mask bits to create shadows, enhancing the 3-D appearance of the object.
Summation of masking

The great plus of masking is that it

+ supports animation of all kinds of shapes.

The red minus however is that it

- uses four copy operations to display each single frame.

If the program itself is in command of the animation process, the most effective approach is to combine masking technique with XOR technique. XOR, using just two copy operations for each frame, is applied to rapid motions, while masking is used for slower motions and static but changing objects.

A call for sprites

One of the main problems with the animation techniques displayed as far, is that real 3-D world is forced into a 2-D flat representation. This is OK for movies and TV, because these media are just reflections of the real world, but the computer animator is trying to produce it with help of the computer. I wish and believe we will see real 3-D video memories in the near future.

A step toward this goal is the EGA/VGA construction of video memory using four color plans, which we can address separately. As we shall see later, we can use these planes to introduce depth in the animation.

One popular technology on dedicated game computers like Amiga and Atari is the so called sprite technology. A sprite is a separate rectangular video memory, or frame buffer, which is merged onto the normal screen video memory during the frame scanning process. The displayed screen image shows the content of the full screen video memory and in front of this the sprite memory. The sprite can be overlaid at any screen position, without copying of any data. The mixing takes place in the video controller, not in memory. To move an object, the first step is to put it in the sprite memory, and then you just change some registers containing the x and y position of the sprite, both a simple and extremely fast operation.

Game computers often have several separate sprite memories that may overlap and hide each other like cards in a card deck. You can give different depth priorities to move sprites from the background to the foreground and vice versa.

For some reason, we don't have sprites on the PC's today and that's why many computers are spending their time just managing animation of the mouse cursor. To be fair, the new IBM XGA has one sprite buffer of 64x64 pixels intended for mouse cursors - but now it's 1992 and we should expect to see better things.
Specialized animation technologies

We're now moving into more specialized and hardware adapted animation technologies, but we try to become not too hardware specific or dependent. So most of the methods described work on different platforms.

Increased speed with byte alignment

Many popular video adapters use a *bit* oriented representation of pixels. This is true for old CGA, Hercules but also EGA and VGA. In fact, the latter two have four bit planes for each pixel, so each pixel is represented by four bits, not one, but the data is always manipulated bitwise in each separate plan.

Other popular adapters as MCGA (or VGA in 256 colors 320x200 resolution), 8514/A and XGA also have *byte* oriented modes, i.e., each pixel is represented by one byte or eight bits. Soon we will see adapters with two, three and perhaps four-byte pixel representations.

All popular video cards are organized row by row with the uppermost row on the lowest memory address. Bit oriented adapters put eight bits into one byte in the x direction. So the 640 pixels per row fit into 640/8 = 80 bytes. The 80x86 processors are not especially good on manipulating individual bits, instead they are devoted to bytes and words. When copying a bitmapped image into video memory we usually copy full bytes or even words, not individual bits, because that would be too slow.

This is OK if the image is copied to x positions starting at byte addresses (i.e., x = 0, 8, 16, 24, etc.). If we, however, copy the image to, say x = 3, than we have to shift every image bit three steps to the right before copying the information. The 80x86 has shift instructions, so this is not a problem (we can even use multiplication or division with 2, 4, 8, etc.), but the problem is even more complicated. If we shift the bits in the image first byte three steps to the right, than we get three outshifted bits, which have to be merged with shifted bits in the second byte, and so on. There is still one problem; in the shifted first byte the three leftmost bits are unused and the background should not be modified here. This is also true with the last byte, which contains three new shifted bits and five unused bits that should not affect the background. So the first and last byte need special treatment.

Sounds complicated? As a *user* of graphic libraries, for instance BGI, you will not need to dwell into this mess any further. All problems are taken care of by the library - but you should know of the *time* aspect. Shift copy is much more complicated than straight byte copy and therefore has a speed penalty. So please avoid shifting during animation.

There are two ways to avoid shifting. The first one is to position objects in the x direction only on byte boundaries (i.e., x = 0, 8, 16, 24, etc.). You should also try to set the image width to multiples of eight (i.e., w = 8, 16, 24, etc.), so you don't need special care for the rightmost bits. This approach is acceptable when animating rapid motion, but will look jerky if the motion is slower.
The second one is to store the shifted object images in memory. Then you need eight shifted versions of each frame and you still have to take care of background protection at the right and left edges. You select the image to copy from the remainder of x MOD 8. Also note that the mask must exist in eight shifted variations - but it is still faster than shifting. This method is often used to draw mouse cursors.

Using byte oriented graphic modes makes all shift and protection problems to disappear. There are no restrictions on x positions and you never have to protect left and right borders.

Palette animation - true magic

The animation technique we will now describe is restricted to palette oriented adapters and graphic modes. On the PC it can be used with EGA and still better with VGA, MCGA (= VGA 256 color 320x200), SuperVGA, XGA and 8514/A. It cannot be used with CGA, Hercules or true RGB color 16 bits or 24-bits modes.

The image displayed on the monitor is not fully given by the bit pattern in video memory. The video controller is doing a good job to translate the bit information of each pixel to a true RGB (red-green-blue) color. Memory is a scarce resource and thus the video adapter only store color indices in video memory and use a look-up table, called the color palette, to translate a specific color index to a real color. With the arising 16 and 24-bits technologies this will be changed and we will store RGB values directly in video memory.

As an example, the VGA adapter in the standard 16 color 640x480 pixel mode, use four bits to represent each pixel, which equals to \(2^4 = 16\) combinations or color indices. The default VGA palette translates these color indices to RGB colors as follows:

<table>
<thead>
<tr>
<th>Video memory color index</th>
<th>Palette R, G, B</th>
<th>Screen color</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 0, 0</td>
<td>black</td>
</tr>
<tr>
<td>1</td>
<td>0, 0,42</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>0,42, 0</td>
<td>green</td>
</tr>
<tr>
<td>3</td>
<td>0,42,42</td>
<td>cyan</td>
</tr>
<tr>
<td>4</td>
<td>42, 0, 0</td>
<td>red</td>
</tr>
<tr>
<td>5</td>
<td>42, 0,42</td>
<td>magenta</td>
</tr>
<tr>
<td>6</td>
<td>42,21, 0</td>
<td>brown</td>
</tr>
<tr>
<td>7</td>
<td>42,42,42</td>
<td>lightGray</td>
</tr>
<tr>
<td>8</td>
<td>21,21,21</td>
<td>darkGray</td>
</tr>
<tr>
<td>9</td>
<td>21,21,63</td>
<td>lightBlue</td>
</tr>
<tr>
<td>10</td>
<td>21,63,21</td>
<td>lightGreen</td>
</tr>
<tr>
<td>11</td>
<td>21,63,63</td>
<td>lightCyan</td>
</tr>
<tr>
<td>12</td>
<td>63,21,21</td>
<td>lightRed</td>
</tr>
<tr>
<td>13</td>
<td>63,21,63</td>
<td>lightMagenta</td>
</tr>
<tr>
<td>14</td>
<td>63,63,21</td>
<td>yellow</td>
</tr>
<tr>
<td>15</td>
<td>63,63,63</td>
<td>white</td>
</tr>
</tbody>
</table>

The color intensity of each primary color is restricted to the interval 0..63, that is 64 intensity levels. This makes \(64^3 = 262,144\) RGB combinations, but we can only use sixteen of these values simultaneously, namely those stored in the current palette.

As a comparison let’s look at the byte oriented adapters and modes (MCGA, VGA 256}
colors 320x200, SuperVGA, XGA and 8514/A). They all use eight bits to represent a pixel and this gives us $2^8 = 256$ color indices. The RGB intensity levels are restricted to 64, so still we have a total of $64^3 = 262,144$ RGB colors, but now we can display 256 colors at the same time.

The RGB colors in the palette may be changed any time without any effect on the content in video memory. This capacity may be used to produce color changes, color enhancements, fading of images, etc. But how do we use palettes for animations, which is our prime interest?
Palette technique for cyclic animations

We will use VGA as a reference, but the technique now described works equally well with other palette oriented adapters and modes.

A VGA pixel is represented by four bits and each bit may be off or on. We may look at a VGA image as holding four separate monochrome images. Using palette manipulations as our magic wand, we can make these four images appear and disappear on the screen at our will. By changing the palette according to a cyclic scheme these four images will act like frames and turn up in an animation sequence, without changing a single bit in the video memory!

We will number our four images as frame 0, 1, 2 and 3. All frames use color index 0 to represent background colors (say black) and color index 1 (image 0), 2 (image 1), 4 (image 2) and 8 (image 3) to represent the foreground color (say white). A table may clarify:

<table>
<thead>
<tr>
<th>Color index</th>
<th>Bits</th>
<th>Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 2 1 0</td>
<td>frame 3</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>black</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>black</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>black</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>black</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td><strong>white</strong></td>
</tr>
</tbody>
</table>

For color index 1, which corresponds to bit 0 set to on and only used by image 0, the corresponding palette color in palette 0 should be white (or foreground). Palette 1, 2 and 3 should use palette color black, and so on. By selecting palette 0, color index 1 will be displayed as white and the palette colors of image 1, 2 and 3 will be set to black. This makes only frame 0 visible as all other frames are black. By selecting palette 1, only frame 1 will show up, and so on.

This run smoothly if all four image frames are separate and don't overlap each other; but in general we also have to consider overlapping.

If frame 0 (foreground color index 1) overlaps frame 2 (color index 4), we will get a combined color index of $1 + 4 = 5$. Therefore the palette color of color index 5 should be set to white. The next table displays all possible combinations when we take overlapping into regard:
<table>
<thead>
<tr>
<th>Color index</th>
<th>Bits</th>
<th>Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0</td>
<td>black black black black</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1</td>
<td>black black black white</td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>black black white black</td>
</tr>
<tr>
<td></td>
<td>0 0 1 1</td>
<td>black black white white</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>black white black black</td>
</tr>
<tr>
<td></td>
<td>0 1 0 1</td>
<td>black white black white</td>
</tr>
<tr>
<td></td>
<td>0 1 1 0</td>
<td>black white white black</td>
</tr>
<tr>
<td></td>
<td>0 1 1 1</td>
<td>black white white white</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
<td>white black black black</td>
</tr>
<tr>
<td></td>
<td>1 0 0 1</td>
<td>white black black white</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>white black white black</td>
</tr>
<tr>
<td></td>
<td>1 0 1 1</td>
<td>white black white white</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0</td>
<td>white white black black</td>
</tr>
<tr>
<td></td>
<td>1 1 0 1</td>
<td>white white black white</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0</td>
<td>white white white black</td>
</tr>
<tr>
<td></td>
<td>1 1 1 1</td>
<td>white white white white</td>
</tr>
</tbody>
</table>

We may construct a compound image containing four separate monochrome images in this way:

First draw image 0 using color index 0. Next establish drawing mode OR and draw frame 1 using color index 2, frame 2 using color index 4 and frame 3 using color index 8. Using the default palette the new image looks real funny.

Now to the animation part: Let’s assume that the four frames hold a repeated animation sequence: Repeatedly change the palette from palette 0 to 1, 2, 3 and then start it all-over again:
Palette animation

1. Set next palette
2. Repeat from step 1

MultiGraphics:

CONST
  maxColIdx    = (hardware dependent: VGA = 15, MCGA = 255)
  maxFrame    = (hardware dependent: VGA = 3, MCGA = ?)
TYPE
  PaletteColor = {hardware dependent}
  PaletteType  = ARRAY[0..maxColorIndex] OF PaletteColor;
  PaletteArray = ARRAY [0..maxFrame] OF PaletteType;
CONST
  aniPal       : PaletteArray = (...);
  ...
FOR i:= 0 TO maxFrame DO BEGIN
  MG.SetDevicePalette(MG.DevicePalette(@aniPal[i]));
  Delay(dt);
END;

BGI:

CONST
  maxColIdx    = (hardware dependent: VGA = 15, MCGA = 255)
  maxFrame    = (hardware dependent: VGA = 3, MCGA = ?)
TYPE
  PaletteColor = {hardware dependent}
  PaletteType  = ARRAY[0..maxColorIndex] OF PaletteColor;
  PaletteArray = ARRAY [0..maxFrame] OF PaletteType;
CONST
  aniPal       : PaletteArray = (...);
  ...
FOR i:= 0 TO maxFrame DO BEGIN
  SetFullPalette(aniPal[i]); {specific to hardware}
  Delay(dt);
END;

Color index 0 and 15 have identical palette colors in all palettes, namely black and white. We may use this fact to store a static black and white image that will show up with every palette.

The strong point of palette animation is that

+ no copy operations are performed during animation. Animation speed not dependent on image size.
Here is a list of week points

- few image frames in each animation sequence
- a limited number of colors
- best suited to repeated sequences
- somewhat complicated to generate images

The number of animated images is maximized to the number of bits to represent a pixel, that is four images using VGA 640x480 and eight images using MCGA/VGA 320x200, SuperVGA and 8514/A.

Sometimes we are satisfied with just two or three frames in a repeated background animation such as waves on sea, water falls, rain and falling snow, etc. Using two monochrome frames on VGA (not necessarily black & white) we are left with two unused bits, representing two colors, which we can use for other purposes. A total of four colors.

With byte oriented modes we may, for instance, set off four bits to get four palette animated frames, and the remaining four bits to get four more colors. The number of simultaneous colors is thus restricted to the number of bits representing a pixel.

Here is a VGA example showing a palette oriented blue & white sequence and two independent colors, namely green & red:

<table>
<thead>
<tr>
<th>Color index</th>
<th>Bits</th>
<th>Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 2 1 0</td>
<td>frame 1</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>white</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>white</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>blue</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>green</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>red</td>
</tr>
</tbody>
</table>

Note that the background in this example is white.

The remaining color indices represent overlapping images. Of course color index 3, which is created when frame 0 and frame 1 overlaps, has to be blue. Color index 5, 6, 7, 9, 10, etc. open up new interesting possibilities.
3-D with palettes

We may select palette colors for the remaining color indices, which come to use when objects are overlapping, to set the depth precedence of each object. If we want the animated blue & white sequence always to stay in the background and that red objects should be in front of green objects, we have to use the following palettes:

<table>
<thead>
<tr>
<th>Color index</th>
<th>Bits</th>
<th>Palette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame 1</td>
<td>Frame 0</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>white</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>white      blue</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>blue        white</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>blue        blue</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>green       green</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>green       green</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>green       green</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>green       green</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>red         red</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>red         red</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>red         red</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>red         red</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>red         red</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>red         red</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0</td>
<td>red         red</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>red         red</td>
</tr>
</tbody>
</table>

All color indices with bit 3 on will use palette color red. All color indices with bit 2 on, except those with bit 3 on, will use palette color green, etc. This makes red objects always to be in front of everything else and green objects to be in front of the background.

Objects of the same color index can move in front and behind objects of different color indices without any conflicts. By changing the palette we can reverse the order. We can even create fancy worlds where objects move in front and behind each other in a paradoxical way. A computer fiction in class with Eschers bizarre perspective drawing.

Palette animation and palette depth technique can be used jointly with the general animation techniques described before.
A few more special techniques

When we save and restore backgrounds and draw images on plane oriented adapters such as EGA and VGA, we can count on a quadruple increase in speed if we do all copy operations inside the video memory, instead of copying to and from ordinary RAM. That's so because the adapter hardware can copy data in the four bit planes in parallel, instead of copying one bitplane at a time. With a 256 K EGA adapter we can store two full video screens, or frame buffers, but only one of them is visible (starting at address A000H:0000H). The rest of the video memory can be used to store bitmaps and to save and restore backgrounds. On VGA we cannot store a full screen but only 640x339 pixels.

Some video adapters support two separate and switchable frame buffers. By a simple call we can change the scan conversion from one video memory to the other.

With EGA and VGA it is possible to change the start address of the frame scanning process (default A000H:0000H) to another address, for instance the second frame buffer in video memory. We can display a frame in the first buffer and at the same time prepare the next frame using the second buffer. Then, preferably during retracing, we switch the hardware to display the second buffer and start to prepare the next frame in the first buffer. This makes absolutely flicker free graphic animation possible, but can only be used fully on EGA 640x350 and pose some new problems on keeping the two buffers updated in parallel.
Managing object dynamics

The subject of computer animation is not exhausted with our description of basic graphic drawing techniques. Other aspects are how to control object motions, timing aspects, interactions of animated objects, changing shapes, perspectives, etc.

As in case of the graphic techniques described above, there are several methods for dealing with these aspects, all with their pro and con's. Some techniques are well suited for animation of single objects, other's for controlling several objects concurrently.

Moving a single object from A to B

We often need to move a fixed screen object, for example a window or icon, from one position (A) to another (B). The object is moved following a straight line. Here is a general Pascal procedure accomplishing this task:

```pascal
TYPE
  IconType = RECORD
    x, y, w, h : INTEGER;
    iconImg, maskImg, xorImg, backImg: MGImg.Image;
  END;

PROCEDURE MoveIconTo(VAR icon : IconType;
  x, y : INTEGER;
  milliSec : Word;
  n : Word);
BEGIN
  { See ICONMOVE.PAS }
END;
```

Controlling several objects

The simple procedure above only deals with straight motions of a single object. If we have several objects on screen we have to cycle between them and let each move a bit and turn to the next object. Motion is treated incrementally, that is in small portions of time and distance.

As with all kinds of concurrent actions we have three different approaches to handle this task:

1. Polling
2. Event based
3. Multi-tasking
Object polling

This simple method repeatedly asks each object what to do (leave unchanged, move, change frame etc.) and then performs the corresponding action:

```
REPEAT
    IF ChangeObject(1) THEN Animate1...;
    IF ChangeObject(2) THEN Animate2...;
    IF ChangeObject(3) THEN Animate3...;
    ...
UNTIL done;
```

This seems to be a good approach, but in reality it has important problems regarding distribution of time between the objects.

Event handling

The next method is first to send all events to a common event queue and then repeatedly fetch and act on events from this queue.

```
REPEAT
    IF EventsPending THEN BEGIN
        FetchEvent; ActOnEvent;
    END;
UNTIL done;
```

Events may be generated in the background by the PC timer (hardware interrupt), by other objects, by a collision detection mechanism in the animation supervisor, etc..

This is the most widely used a preferred technique for controlling animations.

Multi-tasking

If the objects are acting as separate individuals with no or few connections, we will get a specially simple and attractive animation technique, if we use multi-tasking. The motion of each individual object may be described as a separate process without regard of the other actors involved. We may even use simple procedures as MoveObject above.

The time is distributed between the object processes by a process scheduler. Here is some pseudo code:

```
PROCESS Object1;
BEGIN
    REPEAT
        ...
    UNTIL done;
END;
```
Animation on the PC

```plaintext
PROCESS Object2;
BEGIN
  REPEAT
  ... 
  UNTIL done;
END;

BEGIN
  StartProcess(Object1);
  StartProcess(Object2);
  ...
END.
```

Languages such as Modula-2 and ADA have high-level support for processes. Using Pascal and C/C++ you have to rely on interrupt service routines, hooked to the timer interrupt.
**Executables**

4:21 ANIMAT1.EXE

Driver: VGA.MGA
Images: ANIMAT1.PCX, LG6A.PCX..LG6K.PCX

First displays the four walker silhouettes in sequence.
<Enter> to let the walker really walk.
<-> increase his speed, <-> slows down.
<Esc> to quit.

9:62 LGVIDEO.EXE

Driver: MCGA.MGA
Images: LGA.PCX..LGK.PCX

**Draw-Draw**

A sequence of gray scale images captured by a video camera.
<Space> to get full speed.
<Enter> to repeat the sequence.
<Esc> to quit.

13:82 SWAN1.EXE

Driver: VGA.MGA
Images: STHLM.PCX, SWAN31.PCX..SWAN33.PCX

**Draw-Draw**

Three images making up a sequence of a flying swan. First displayed in "slow-motion".
<Space> to toggle "slow-motion" on/off.
<-> to increase the wing-beat speed, <-> to slow down.
<Esc> to quit.
Animation on the PC

17:102 SWAN2.EXE

Driver: VGA.MGA
Images: STHLM.PCX, STHLMTEXT.PCX, SWAN31.PCX..SWAN33.PCX

Draw-Erase

The swan is now flying across the screen.
\(<Space>\) to toggle "slow-motion" on/off.
\(<+>\) to increase the speed, \(<->\) to slow down.
\(<Enter>\) for next "scene" in presentation.
\(<Esc>\) to quit. Last scene is when the swan is flying in front of the city tower (and make some damage to it).

20:112 SWAN3.EXE

Driver: VGA.MGA
Images: STHLM.PCX, SWAN31.PCX..SWAN33.PCX

Draw & Erase

\(<Space>\) to toggle "slow-motion" on/off.
\(<+>\) to increase the speed, \(<->\) to slow down.
\(<Enter>\) for next "scene" in presentation.
\(<Esc>\) to quit. Last scene is when the swan is flying in front of the city tower (and make some damage to it).

26:152 SWAN4.EXE

Driver: VGA.MGA
Images: STHLM.PCX, SWAN31.PCX..SWAN33.PCX

Save - Draw - Restore

\(<Space>\) to toggle "slow-motion" on/off.
\(<+>\) to increase the speed, \(<->\) to slow down.
\(<Enter>\) for next "scene" in presentation.
\(<Esc>\) to quit. Last scene is when the swan is flying in front of the city tower without a frame.
35:222 SWAN5.EXE
Driver: VGA.MGA
Images: STHLM.PCX, SWAN31.PCX..SWAN33.PCX

XOR-draw

<Space> to toggle "slow-motion" on/off.
<-> to increase the speed, <-> to slow down.
<Enter> for next "scene" in presentation.
<Esc> to quit. Last scene is when the swan is flying in front of the city tower without color mixing.

37:231 XORDEMO.EXE
Driver: VGA.MGA
Images: WINDOW.PCX
Fonts: TRIP.CHR

XOR-draw

Four scenes:

1.XOR Rubber Band Line Drawing
Press left mouse button and keep pressed to draw rubber band line. Release button to accept line. <Enter> or right button for next scene.

2.XOR Rubber Band Rectangle Drawing
Press left mouse button and keep pressed to draw rubber band rectangle. Release button to accept rectangle. <Enter> or right button for next scene.

3.XOR Window Move
Move mouse cursor inside window, press left button and keep pressed to move window. Release button to put window in new position. <Enter> or right button for next scene.

4.XOR Cursors
Move cursor across text. Press <Enter> or right button for next scene with color bar. Note the effects on cursor color depending on background color.

<Esc> to quit.
Animation on the PC

43:267 SWAN6.EXE

Driver: VGA.MGA
Images: STHLM.PCX, SWAN31.PCX..SWAN33.PCX, SWAN41.PCX..SWAN43.PCX

Save - AND mask - OR draw - Restore

<Space> to toggle "slow-motion" on/off.
<+> to increase the speed, <-> to slow down.
<Enter> for next "scene" in presentation.
<Esc> to quit. Last scene is when the swan is flying in front of the city tower without a surrounding frame.

50:312 SNOWVGA.EXE

Driver: VGA.MGA
Images: SNOW.PCX, SNOGUBBE.PCX

Palette animation VGA 16 colors mode

First scene shows the cyclic background pattern with default color palette. Next scene starts the palette animation. Last scene also shows a static snowman. Use <Enter> to shift scene and <Esc> to quit.

52:331 SWAN7.EXE

Driver: VGA.MGA
Images: STHLM.PCX, SWAN31.PCX..SWAN33.PCX,

Palette animation

<Space> to toggle "slow-motion" on/off.
<+> to increase the speed, <-> to slow down.
<Enter> for next "scene" in presentation.
<Esc> to quit.

56:346 SNOW256.EXE

Driver: MCGA.MGA
Images: SNOWBYTE.PCX
Palette animation MCGA 256 colors mode

First scene shows the cyclic background pattern with default color palette. Next scene starts the palette animation. Note the "in front of" and "behind" relations of objects. Use <Enter> to shift scene. <Esc> to quit.

57:347 WALKER.EXE

Driver: VGA.MGA
Images: LG6A.PCX..LG6K.PCX

Palette & standard animation techniques VGA 16 colors mode

Three men walk across the screen. Note different speed and depth relations. Use <+> to increase speed, <-> to slow down. <Esc> to quit.

60:361 ICONMOVE.EXE

Driver: VGA.MGA
Images: SWFLAG.PCX,BALLON.PCX,BALLON0.PCX,
       VOLVO.PCX,VOLVO0.PCX,
       DALA.PCX,DALAMASK.PCX,DALAOUTL.PCX
Fonts : SANS12.CBS

Four icons that can be moved following strait lines. Position the mouse cursor at the icon you want to move, press left mouse button (fast move) or right button (slow move), keep pressed and release button at position you want the object to be moved to. Each icon is moved using a slightly different technique.

Please do no not position icons above each other - this simple program can't manage the situation! <Esc> to quit.
Index

8514/A
Clipping ........................................... D6-9
Colors ........................................... D6-7
Line style ....................................... D6-4
Optimizing ..................................... D6-4
Pattern............................................... D6-5
Special screens ............................. D6-3
Virtual screens .............................. D6-2
Visible screen ............................... D6-2
World coordinates ........................... 21-5, 22-5

A
Activate device ................................ 5-21
Adapter
Detect SuperVGA............................ D11-9
Add character
AddChar......................................... 15-22
Add symbol
AddSymbol................................ 14-24
AddChar
Add character ............................... 15-22
AddDisplayObject
MGDO............................................ 25-11
AddImageMask ................................ 20-5
MGImg........................................... 20-14
AddSymbol
Add symbol.................................. 14-24
AdjustCharPos
MGIO................................ .......... 23-17
AdjustWin
MG.............................................. 7-10
Alarm
MGClock ......................................... 28-8
Set repeating ................................. 28-13
Set single ..................................... 28-12
Stop repeating ................................. 28-13
Switch off one-time ......................... 28-13
AlarmSet
MGClock ......................................... 28-9
allEventGens

MGEv.............................................. 27-7
allEvents
MGEv.............................................. 27-7
allKbdEvents
MGKbd........................................... 29-3
allMouseEvents
MGMouse....................................... 30-11
allSymbols
MG............................................... 14-13
AnaClockLook
MGClock......................................... 28-10
AnaClockShape
MGClock......................................... 28-10
Analog clock ................................. 28-18
AND
Drawing mode......................... 9-9
Angle
Current ........................................ 22-20
Get current .................................. 9-15
Set ............................................. 22-19
Set current ................................... 9-14
Summary ....................................... 9-3
Turn ........................................... 9-14, 22-20
AppendDisplayElement
Display list ..................................... 24-16
AppendFromDisplayFile
MGDL............................................ 24-24
AppendToDisplayFile
MGDL............................................ 24-23
Arc
Draw ........................................... 11-17, 21-23, 22-26
Area
Fill irregular ......................... 12-13
Pattern fill ................................... 12-13

MultiGraphics & Borland Pascal/Turbo Pascal   DATABITEN AB
Aspect ratio
  Current ........................................... 5-23
  MG .................................................. 5-5
  Set .................................................. 5-23

Attribute
  Character ........................................... 23-3

Attributes
  Position ............................................... 23-4

AttrStrWidth
  MGIO .................................................. 23-20

Auto detection
  Override ............................................. 5-4, 5-14

AutoDevice ............................................. 5-14

Automatic loading
  Driver .................................................. 5-14

AutoVM
  Automatic virtual memory .......... B-3

Average width
  Font .................................................. 23-16

B

Background color ........................................... 8-6
  Read .................................................. 8-24

Background colors
  Current ............................................. 8-23
  Match .................................................. 8-23
  Set .................................................. 8-23
  Similar ............................................. 8-20

Base line
  Attribute character ....................... 23-20
  Character ........................................... 15-26
  Characters ........................................... 15-6
  Fonts .................................................. 21-31, 22-36

Baud rate
  Set setting ........................................... 26-8

Beep
  MGIO .................................................. 23-33
  Set .................................................. 23-32
  Sound .................................................. 23-33
  Used? .................................................. 23-32

Bezier
  MGW .................................................. 22-8

Bezier curve
  Draw .................................................. 11-23
  Properties ........................................... 11-2

Bézier curve
  Draw .................................................. 21-25, 22-29
  Fill .................................................. 12-12
  Filling of ........................................... 21-29, 22-33

Bézier fitting
  MGBez .................................................. 31-1, 31-2, 31-3

Bézier font ............................................. 15-2, 15-5

Bézier symbols ............................................ 14-1, 14-4

BGI emulation ............................................ D-1

Bit block
  Draw .................................................. 13-7
  MG .................................................. 13-3
  Read .................................................. 13-9

Bitmapped font ............................................. 15-2, 15-3

Bitmapped fonts
  Conversion to ........................................... 15-15
  Convert to ............................................ 15-6

Bitmapped symbols ........................................... 14-1, 14-3
  Convert to ............................................ 14-5, 14-18

bitPat8
  MGPat .................................................. 17-4

Black-on-white
  Color selection ........................................... 8-7
  Current .................................................. 8-19
  Set .................................................. 8-19

Bold characters
  MGIO .................................................. 23-3

Border color
  EGA .................................................. D3-7
  VGA .................................................. D4-7

Borland
  Emulation BGI ........................................ D-1

Box
  Draw .................................................. 11-15, 21-21, 22-25
  BP.EXE ................................................ 1-4

bufferPtr
  MG .................................................. 2-4

bufferSize
  Size of buffer ........................................... 2-4

Button
  MGMouse ............................................. 30-13
  Button pressing ........................................ 30-35
  Buttons
    Mouse .................................................. 30-38
  ButtonSet
    MGMouse ............................................. 30-13

C

CallStrEditor ............................................. 23-8
  MGIO .................................................. 23-50

Canon Laser
  getDeviceModeInfo .................................. D14-9
  getDeviceSettings .................................. D14-9
MultiGraphics

Line style .................................. D14-3
MGCanon .................................. D14-1
Printer fonts .............................. D14-4
setDensity ................................ D14-8
SetDeviceCall........................... D14-7
The output area......................... D14-7
Virtual screens .......................... D14-3
Visible screen ........................... D14-2

cap
MGIW.......................................... 21-6
MGW.......................................... 22-8

Capability
Clock............................................ 28-4
Exists ........................................... 7-18
Increase...............................6-13, 7-14
MG................................................. 7-7
MGIW.......................................... 21-6
MGW.......................................... 22-8
Screen ............................................ 6-6
Text I/O ....................................... 23-2
Variable for..............................23-15
Window ......................................... 7-4

CBL
Bitmapped large font................... 15-7

CBS
Bitmapped small font .................. 15-7

CD
Device font .................................. 15-7

CGA
Colors.......................................... D1-5
Driver.......................................... D1-1
Line style .................................... D1-3
Optimizing.................................. D1-2
Patterns ....................................... D1-3
Virtual screens .......................... D1-2
Visible screen ........................... D1-1

Character
Add ............................................ 15-22
Delete......................................... 15-21
First in font .............................. 15-27
Height ........................................ 23-20
Last in font .............................. 15-27
Width ........................................ 23-19

Character attribute
Position ...................................... 23-25, 23-26

Character attributes
Get ............................................. 23-24
MGIO........................................... 23-3
Set ............................................. 23-24

Character data
GetCharData.................................. 15-17
GetCharDataPtr......................... 15-19

Character height
Attribute ........................................ 23-20
Font ........................................... 15-27
Fonts......................................... 21-32, 22-36

Character information ................. 15-6

Character position
Adjust........................................ 23-17

Character set
HP LaserJet................................ D8-10

Character width
Attribute ........................................ 23-19
Fonts......................................... 15-26, 21-31, 22-35

Characters
Proportional .................................. 15-6
Read ........................................... 23-55
Write ........................................... 23-39

CharAttributePos
Attributposition.......................... 23-11

CheckNextEvent
MGEv......................................... 27-13

Circle
Pattern fill .................................. 12-8

Circle sector
Draw................................. 11-19, 21-23, 22-27
Fill................................. 12-9, 21-27, 22-31

Circles
Draw................................. 11-17, 21-22, 22-26
Pattern fill.............................. 21-27, 22-31

Clear
Line ........................................... 23-21
Window ....................................... 23-21

ClearDisplayList
MGDL........................................24-15

Clipping
8514/A......................................... D6-9
Summary ...................................... 9-3
Viewport ...................................... 7-5

Clock
Analog ........................................ 28-10
Draw Clock-face.......................... 28-18
Draw hands ................................ 28-19
Hand type .................................. 28-18
Stop ........................................... 28-12

Clock event
Data types .................................... 28-9

Clock events
Start........................................... 28-11

Clock ticks
Per second ................................... 2-2
Clock-face
  Appearance ........................................... 28-10
  Digital .................................................. 28-16
  Draw ........................................................ 28-18
ClockAtTime
  MGClock ................................................... 28-15
ClockEvent
  MGClock ................................................... 28-9
clockGen
  MGEv ........................................................ 27-7
ClockNow
  MGClock ................................................... 28-15
ClockToTime
  MGClock ................................................... 28-14
Clone
  Image ........................................................ 20-14
Clone screen .................................................. 6-10
CloneImage
  MGImg ....................................................... 20-14
CloneScr
  MG .......................................................... 6-10
Close device ................................................... 5-15
CloseDevice
  MG .......................................................... 5-15
CloseDISK
  MGOut ......................................................... 26-10
ClrEol
  MGIO .......................................................... 23-21
ClrIOWin
  MGIO .......................................................... 23-21
ClrWin
  MG .......................................................... 7-22
COL
  Large outline font ......................................... 15-7
  Small outline font ......................................... 15-7
Color
  Color index .................................................. 8-5
  Current background's ....................................... 8-23
  Current foreground's ....................................... 8-21
  Get RGB colors .............................................. 8-18
  Match background's ......................................... 8-23
  Match foreground's ......................................... 8-22
  Physical meaning of ....................................... 8-2
  RGB value ..................................................... 8-2
  Set background's ............................................ 8-23
  Set foreground's ............................................. 8-21
  Set palette colors .......................................... 8-17
Color index
  Background ................................................... 8-20
  Current maximum index .................................... 8-18
Foreground ..................................................... 8-20
Max ............................................................. 8-11
Summary ....................................................... 8-5
Color intensity
  DeskJet ....................................................... D13-12
Color mask
  Get current ................................................... 9-17
  Max in image ............................................... 20-28
  Maximum for screen ........................................ 6-17
  Set current ................................................... 9-16
  Summary ....................................................... 9-5
Color plane
  Maximum ....................................................... 6-16
Color selection
  Palette ....................................................... 8-4
  True color ..................................................... 8-3
Color tone
  HSV ............................................................. 18-5
Colors
  DeskJet ....................................................... D13-3
  MG ............................................................. 8-1
  PaintJet ....................................................... D12-3
ColorTone ...................................................... 18-5
Column
  Read current ............................................... 23-18
  Column,row .................................................. 23-17
Columns
  Number of in window ....................................... 23-16
COM-ports ......................................................... 26-8
COM1
  MGOut .......................................................... 26-8
COM2
  MGOut .......................................................... 26-8
Compatibility
  Image/screen .................................................. 20-22
Compiler
  Configuration ................................................ 1-3
  Drawing mode .................................................. 9-9
Compression
  DeskJet ....................................................... D13-11
  PaintJet ....................................................... D12-12
CompressionSet
  MGDJet ......................................................... D13-12
  MGPJet ......................................................... D12-12
Contour tracing
  Test function ................................................. 10-2
MultiGraphics

Control point
  Bezier curves ....................... 11-2
Controlling keys
  MGIO .................................. 23-9
Conversion
  Fonts ................................... 15-6
  Of symbols .......................... 14-5
Convert
  Font .................................. 15-15
  Symbols .............................. 14-18
Coordinates
  Get current ........................... 9-12
  MGIW ................................ 21-1
  MGW .................................. 22-1
  Read .................................. 21-16, 22-18
  Set ................................... 21-15, 22-16
  Set current .......................... 9-11
Copy
  Between different devices ........... 13-2
  Graphics ............................. 13-1
  Rectangle area ... 13-4, 21-29, 22-34
  Viewport ............................ 13-6
CopyRect
  MG ................................... 13-4
  MGIW ................................ 21-29
  MGW .................................. 22-34
CopyView
  MG ................................... 13-6
CreateBitmapFont
  MG ................................... 15-15
CreateBitmapSymbols
  MG ................................... 14-18
CreateDisplayFile
  MGDL .................................. 24-22
CreateDisplayList
  MGDL .................................. 24-15
CreateDisplayObjects
  MGDO .................................. 25-8
CreateFont
  Create font .......................... 15-11
CreateImage
  MGImg ................................ 20-7
  MGIImg ................................ 20-8
CreateMonoVirtScr
  MG ................................... 6-9
CreateRGBPalette
  MG ................................... 8-14
CreateSpecScr
  MG ................................... 6-10
CreateStrEditor ..................... 23-8
  MGIO ................................ 23-49

CreateSymbols
  Create symbol library .............. 14-14
CreateVirtScr
  MG ................................... 6-7
CreateWin
  MG ................................... 7-8
  MGIO ................................ 23-15
  MGIW ................................ 21-7
  MGW .................................. 22-9
cross ................................. 17-2

CRT
  And MGIO ............................. 23-5
  Detection ............................ 5-9, 5-10

CSL
  Large stroke font ................... 15-7

CSS
  Small stroke font .................... 15-7

CurAspect
  MG ................................... 5-23
CurAttrCharBaseline
  MGIO ................................ 23-20
CurAttrCharHeight
  MGIO ................................ 23-20
CurAttrCharWidth
  MGIO ................................ 23-19
CurBackground
  MG ................................... 8-23
CurBeep
  MGIO ................................ 23-32
CurBlackOnWhite
  MG ................................... 8-19
CurCharBaseline
  MG ................................... 8-19
  MGIW ................................ 15-26
  MGW .................................. 22-36
CurCharHeight
  MG ................................... 15-26
  MGIW ................................ 21-31
  MGW .................................. 22-35
CurCharWidth
  MG ................................... 15-26
  MGIW ................................ 21-31
  MGW .................................. 22-35
CurCol
  MGIO ................................ 23-18
CurColorMask
  MG ................................... 9-17
CurCursor
  MGIO ................................ 23-28
CurCursorBlink
MGIO.......................... 23-31
CurCursorColor
MGIO.......................... 23-31
CurDevice
MG......................... 5-21
CurDeviceFamily
MG......................... 5-22
CurDeviceMode
MG......................... 5-22
CurDevicePalette
MG......................... 8-16
CurDir
MG.............................. 9-15
MGW ......................... 22-20
CurDisplayList
MG.............................. 24-13
CurDisplayListPos
MGDL......................... 24-22
CurDrawMode
MG.............................. 9-18
CurEndPointSkip
MG.............................. 11-8
CurErrorWin
MG............................... 3-5
CurEventIntrVec
MGEv ......................... 27-12
CurFirstChar
MG.............................. 15-27
CurFirstSymbol
MG.............................. 15-27
CurFont
MG.............................. 14-30
CurFontType
MG.............................. 15-25
CurForeground
MG.............................. 8-21
CurLastChar
MG.............................. 15-27
CurLastSymbol
MG.............................. 15-27
CurLineStyle
MG.............................. 11-5
CurLineStylePos
MG.............................. 11-6
CurLineWidth
MG.............................. 11-8
CurMaxColorMask
MG.............................. 6-17
CurMaxDeviceColor
MG.............................. 8-18
CurMaxEvents
MGEv......................... 27-13
CurMaxPaletteIndex
MG.............................. 8-13
CurMaxPlane
MG.............................. 6-11
CurMouseDev
MGMouse...................... 30-22
CurMouseScr
MGMouse...................... 30-22
CurMouseWin
MGMouse...................... 30-22
CurOrientation
MG.............................. 5-22
CurPattern
MG.............................. 5-22
CurPointDeviceColor
MG.............................. 10-4
CurPointerBG
MGMouse...................... 30-31
CurPointerFG
MGMouse...................... 30-30
CurPointerRedrawTime........ 30-34
CurPointerType
MGMouse...................... 30-30
CurProposal
MGIO.......................... 23-33
CurProtectLineStyle
MG.............................. 11-7
Current device.................. 5-5, 5-21
Set.......................... 5-21
Current palette.................. 8-5
Current screen.................. 6-3
Set visible.................. 6-15
Current window.................. 7-4
CurRow
MGIO.......................... 23-18
CurScr
MG.............................. 6-14
CurScrHeight
MG.............................. 6-17
CurScrollMode
MGIO.......................... 23-23
CurScrType
MG.............................. 6-16
CurScrWidth
MG.............................. 6-16
CurShowCursor
MGIO ......................................... 23-30

Cursor
Blinking? ................................... 23-31
Current color.............................. 23-31
Draw .......................................... 23-30
Hide ........................................... 23-29
Is shown? ................................... 23-30
Read type ................................... 23-28
Set blinking ................................ 23-31
Set color.................................... 23-30
Set type ...................................... 23-28
Show .......................................... 23-29

CurSymbolHeight
MG............................................. 14-29
MGIW........................................ 21-30
MGW ......................................... 22-35

CurSymbols
MG............................................. 14-28

CurSymbolsType
MG............................................. 14-28

CurSymbolWidth
MG............................................. 14-29
MGIW........................................ 21-30
MGW ......................................... 22-34

CurUnderline
MGIO......................................... 23-27

CurWin
MG............................................... 7-17

CurWinCols
MGIO......................................... 23-16

CurWinHandle
MG............................................... 7-19

CurWinHeight
MG............................................... 7-20

CurWinRows
MGIO......................................... 23-16

CurWinWidth
MG............................................... 7-20

CurWinX
MG............................................... 7-19

CurWinY
MG............................................... 7-20

CurX
MG............................................. 9-12
MGIW........................................ 21-16
MGW ......................................... 22-18

CurY
MG............................................. 9-13
MGIW........................................ 21-17

MGW ........................................... 22-18

CZL
Large Bézier font ........................... 15-7

D
dashed ........................................... 17-2
dashedDotted .................................. 17-2

Data
As reference ................................. 24-8
As value ....................................... 24-8

Deallocate image ........................... 20-16

Debugging
MGDebug ...................................... E-1

degrees ........................................ 9-3

Decimal format
WrFixReal .................................... 23-43

Default font ................................ 15-1, 15-2
8514/A ........................................... D6-5
CGA ............................................. D1-4
EGA ............................................. D3-5
Hercules ..................................... D5-4
MCGA .......................................... D2-3
Set ............................................. 15-24
SVGA ........................................... D11-5
Toshiba ....................................... D10-3
VGA ............................................. D4-5

Default palette ................................ 8-5
Set ................................................ 8-11

Default symbols
MG............................................. 14-5
Set ............................................. 14-27

Delay
Delay .......................................... 4-4
Timer delay .................................... 4-4

Delete character
DeleteChar ..................................... 15-21

Delete symbol
DeleteSymbol ............................... 14-24

DeleteChar
Delete a character .......................... 15-21

DeleteDisplayElement
Erase element .................................. 24-6
MGDL ........................................... 24-18

DeleteDisplayObject
MGDO ......................................... 25-12

DeleteSymbol
Delete a symbol ............................. 14-24
dens16 .......................................... 17-4
dens8 ........................................... 17-4
Density
  Data type.................................. 17-3
Depletion
  DeskJet.................................... D13-12
DepletionSet
  MGJet...................................... D13-12
DeskJet
  Color depletion.......................... D13-12
  Color graphics.......................... D13-7
  Colors.................................... D13-3, D13-9
  Compression............................. D13-11
  getDeviceModeInfo..................... D13-14
  getDeviceSettings..................... D13-15
  Line style................................ D13-4
  Multiple passes........................ D13-12
  Pattern................................... D13-4
  Printer fonts............................ D13-5
  setDensity................................ D13-11
  setSize.................................. D13-10
  Virtual screens........................ D13-3
  Visible screen......................... D13-2
DeskJet printers
  Driver..................................... D13-1
Detect
  Basic driver................................ 5-11
  Drivers................................... 5-11
DetectDevice
  MG.......................................... 5-9
DetectDeviceDriver
  MG.......................................... 5-10
detectDeviceMode
  Detect SuperVGA........................ D11-9
Detection
  CRT device................................ 5-9
  CRT driver................................ 5-10
DevHeight
  MGIW...................................... 21-13
  MGW....................................... 22-14
Device
  Activate.................................. 5-22
  Close..................................... 5-15
  Current................................... 5-5, 5-21
  Current aspect ratio................... 5-23
  Current mode............................ 5-22
  Detect basic............................. 5-11
  Detect more drivers.................... 5-11
  Family number........................... 5-22
  For mouse............................... 30-22
  Load driver............................. 5-12
  MG......................................... 5-6
  Orientation............................. 5-22
  Read information....................... 5-20
  Set aspect ratio....................... 5-23
  Special call............................ 5-18, 5-19
Device color
  Allowed.................................... 6-4
  Current maximum of.................... 8-18
  Foreground................................ 8-20
Device colors
  Matching background.................... 8-20
  Device driver........................... 5-2
  Device font............................. 15-2
Device mode
  Set........................................ 5-16
DeviceCall
  MG.......................................... 5-19
DeviceInfo
  MG.......................................... 5-8, 5-16
DeviceInitProc
  MG.......................................... 5-7
DeviceModeInfo
  MG.......................................... 5-8
DeviceOutput
  Example................................... 26-2
  MGOut..................................... 26-4
DeviceOutputProc
  MGOutput................................ 5-7
Devices
  Family.................................... 5-2
  DeviceType................................ 5-7
DevWidth
  MGIW...................................... 21-12
  MGW....................................... 22-14
DevX
  MGIW...................................... 21-11
  MGW....................................... 22-13
DevY
  MGIW...................................... 21-12
  MGW....................................... 22-14
  diagCross................................ 17-2
  diagDown................................. 17-2
  diagUp.................................... 17-2
DigiClockLook
  MGClock.................................. 28-10
Digital clocks
  Draw......................................... 28-16
  Draw digits.............................. 28-17
Digits
  Digital clocks........................... 28-17
Direction
  Current.................................... 22-20
  Get current............................. 9-15
| Set .............................................. | 22-19 |
| Set current.................................... | 9-14 |
| Summary........................................ | 9-3  |
| Turn ..................................9-14, 22-20 |
| DISK                               | |
| Close output file ........................ | 26-10 |
| MGOut ....................................... | 26-9 |
| Printing to .................................. | 26-10 |
| Display file                          | |
| Display list.................................. | 24-7 |
| Display list                          | |
| Append elements                       | 24-16 |
| Clear empty                           | 24-15 |
| Current                               | 24-13 |
| Current position                      | 24-22 |
| Data types                            | 24-11 |
| Delete                                | 24-16 |
| Draw current element                   | 24-19 |
| Draw display list                     | 24-19 |
| Element pointer                       | 24-18 |
| Empty                                 | 24-14 |
| Erase element                         | 24-18 |
| File type                             | 24-2  |
| Final position                        | 24-22 |
| First position                        | 24-21 |
| Insert element                        | 24-18 |
| Insert from file                      | 24-25 |
| Inspect                               | 24-6  |
| Next position                         | 24-20 |
| Number of elements                    | 24-16 |
| Playback                              | 24-4  |
| Portability                           | 24-9  |
| Previous position                     | 24-21 |
| Processing                            | 24-6  |
| Read from file                        | 24-24 |
| Record in                             | 24-12 |
| Recording                             | 24-3  |
| Reset loading routine                 | 24-23 |
| Save on disk                          | 24-7  |
| Set first position                    | 24-21 |
| Set position                          | 24-20 |
| Stop recording                        | 24-12 |
| Write to file                         | 24-23 |
| Display list file                     | Create........... | 24-22 |
| Display lists                          | Overview............... | 24-1 |
| Display object                         | Draw .................. | 25-13 |
| Display object file                   | Save .................. | 25-14 |
| Display object library .................. | 25-1 |
| Creating .................................. | 25-2 |
| Save ...................................... | 25-14 |
| Display objects                       | Add.......................... | 25-11 |
| Create library of ...................... | 25-8 |
| Description of ........................... | 25-1 |
| Draw ..................................... | 25-3 |
| Erase .................................. | 25-12 |
| Get information .......................... | 25-9 |
| Load ..................................... | 25-14 |
| Load by name ................................ | 25-15 |
| Max. number of ........................... | 25-6 |
| Name ................................... | 25-10 |
| Object number ............................. | 25-9 |
| Portability ............................... | 25-4 |
| Summary .................................. | 25-1 |
| Versus procedures ....................... | 25-5 |
| DisplayElement ........................... | 24-11 |
| DisplayElementPtr ....................... | 24-11 |
| DisplayElements                       | MGDL...................... | 24-16 |
| DisplayList                           | 24-11 |
| DisplayListOK ............................. | 24-16 |
| DisplayObjectInfo ....................... | 25-9 |
| MGDO...................................... | 25-7 |
| DisplayObjectName ....................... | 25-7 |
| MGDO...................................... | 25-10 |
| DisplayObjectNameStr .................... | 25-7 |
| DisplayObjects                        | MGDO.......................... | 25-7 |
| DisplayObjectSet ........................ | MGDO.......................... | 25-7 |
| Dispose .................................. | 6-13 |
| DisposeDevicePalette .................. | 8-16 |
| MG ......................................... | 8-16 |
| DisposeDisplayList ...................... | 24-16 |
| DisposeDisplayObjects .................. | 25-8 |
| DisposeFont ................................ | 15-14 |
| MG ......................................... | 15-14 |
| DisposeImage ................................ | 20-9 |
| MGBGI ..................................... | 20-9 |

MultiGraphics & Borland Pascal/Turbo Pascal  
DATABITEN AB
Index

MGImg ...................................... 20-16
DisposeRGBPalette
MG ........................................... 8-14
DisposeScr
MG ........................................... 6-13
DisposeStrEditor 23-8
MGIO ....................................... 23-52
DisposeSymbols
MG ........................................... 14-17
DisposeWin
MG ........................................... 7-11
Distance
Displacement 21-16, 22-17
Move ....................................... 9-12
DL
Explanation .................................. 24-8
File type display lists 24-2
DL(R)
Explanation .................................. 24-9
DLR
Explanation .................................. 24-8
Dot density
Canon Laser ................................ D14-8
DeskJet ....................................... D13-11
Epson ......................................... D7-10
HP LaserJet .................................. D8-9
PaintJet ....................................... D12-11
dotted ........................................... 17-2
Double-clicking
Mouse ......................................... 30-3, 30-35
DPIMEM
BP environment ................................ 1-4
Draw
Bit block .................................... 13-7
Image ......................................... 20-21
Part of image ................................ 20-23
DrawAnaClock
MGClock ..................................... 28-18
DrawArc
MG ........................................... 11-17
MGIW ......................................... 21-23
MGW ......................................... 22-26
DrawBitBlock
MG ........................................... 13-7
DrawBox
MG ........................................... 11-15
MGIW ......................................... 21-21
MGW ......................................... 22-25
DrawChar procedure
Read which ................................... 23-38
Set ............................................. 23-37
DrawCharProc
MGIO ......................................... 23-12
DrawCircle
MG ........................................... 11-17
MGIW ......................................... 21-22
MGW ......................................... 22-26
DrawCurPoint
Draw point ................................... 10-3
DrawCursor
MGIO ......................................... 23-10
DrawDigiClock
MGClock ..................................... 23-28
DrawDigits
MGClock ..................................... 28-17
DrawDisplayElement
Draw element .................................. 24-5
MGDL ......................................... 24-19
DrawDisplayList
MGDL ......................................... 24-19
Playback ...................................... 24-5
DrawDisplayObject
MGDO ......................................... 25-13
DrawEllipse
MG ........................................... 11-19
MGIW ......................................... 21-24
MGW ......................................... 22-27
DrawEllipseArc
MG ........................................... 11-21
MGIW ......................................... 21-24
MGW ......................................... 22-28
DrawEllipseSector
MG ........................................... 11-21
MGIW ......................................... 21-25
MGW ......................................... 22-28
DrawHands
MGClock ..................................... 28-19
DrawImage
MGImg ....................................... 20-21
DrawImagePart
MGImg ....................................... 20-21
Drawing
Text ......................................... 21-33, 22-37
Drawing direction
Summary ...................................... 9-3
Drawing mode
Data type ..................................... 9-10
Get current .................................. 9-18
Mouse pointer ................................ 30-6
Set current .................................. 9-17
## MultiGraphics Index

### Drawing modes
- Available ..................................... 9-18
- Summary ...................................... 9-6

### Drawing point
- Symbol......................................... 21-31, 22-35

### Drawing position
- Summary ...................................... 9-1

### Drawing speed
- HPGL plotters .................................... D9-8

### Drawing text
- HG ............................................. 15-29

### DrawLine
- MG ............................................. 11-10
- MGIW ........................................ 21-20
- MGW ......................................... 22-23

### DrawLineRel
- MG ............................................. 11-9
- MGIW ........................................ 21-20
- MGW ......................................... 22-23

### DrawLineTo
- MG ............................................. 11-9
- MGIW ........................................ 21-20
- MGW ......................................... 22-22

### DrawMode
- MG ............................................. 9-10
- Summary ...................................... 9-6

### DrawModeSet
- MG ............................................. 9-10

### DrawPoint
- Draw point ................................... 10-3
- MGIW ........................................ 21-19
- MGW ......................................... 22-22

### DrawPolyBezier
- Bézier curves .................................. 11-3
- Draw Bézier curve11-23, 21-25, 22-29

### DrawPolyline
- MG ............................................. 11-12
- MGIW ........................................ 21-21
- MGW ......................................... 22-24

### DrawRect
- MG ............................................. 11-16
- MGIW ........................................ 21-22
- MGW ......................................... 22-25

### DrawSector
- MG ............................................. 11-19
- MGIW ........................................ 21-23
- MGW ......................................... 22-27

### DrawShortPolyline
- MG ............................................. 11-14

### DrawStr
- MG ............................................. 15-29

### DrawSymbol
- MGIW ........................................ 21-33
- MG ............................................. 22-37

### Dynamic loading
- Driver ........................................... 5-2

### E

#### Editing
- Text ........................................... 23-6

#### Editing modes
- Read ........................................... 23-34
- Set ............................................ 23-34

#### Editing strings
- Basic procedure ......................... 23-46

#### EditModeSet
- MGIO ........................................... 23-12

#### Editor
- Call ............................................. 23-50
- Create ........................................ 23-49

#### EGA
- Colors ........................................ D3-6
- Driver ........................................ D3-1
- Line style ................................... D3-4
- Optimizing .................................. D3-3
- Patterns ...................................... D3-4
- Special screens ......................... D3-2
- Virtual screens ........................... D3-2
- Visible screen ............................. D3-1

#### Elements
- Display list .................................. 24-16

#### Ellipse
- Draw .......................................... 11-19, 21-24, 22-27
- Pattern fill ............................... 12-10, 21-28, 22-32

#### Ellipse sector
- Pattern fill .................................. 12-11

#### Elliptical arc
- Draw .......................................... 11-21, 21-24, 22-28

#### Embossed characters
- MGIIO ......................................... 23-3

#### EMS
- EmsVM ......................................... B-2
- EmsVM
EventGen
  MGEv .......................................... 27-8
EventGenSet
  MGEv .......................................... 27-8
eventHandlerActive
  MGEv .......................................... 27-7
eventHandlerNotActive
  MGEv .......................................... 27-7
EventMatch
  MGEv .......................................... 27-8
Events
  Keyboard ..................................... 29-1
  Maximum number of ........................... 27-13
  MGEv .......................................... 27-1
  Start mouse .................................. 30-18
  Stop keyboard ................................ 29-6
  Stop mouse ................................... 30-19
  Supervising .................................... E-6
EventServerProc
  MGEv .......................................... 27-5, 27-9
EventsPending
  MGEv .......................................... 27-13
Expanded memory
  EmsVM .......................................... B-2
Exponent format
  WrReal .......................................... 23-43
Extended memory
  XMSVM .......................................... B-2

F
Face of clock .................................. 28-18
Family
  Devices .......................................... 5-2
Family number
  Device's.......................................... 5-22
Fast positioning ............................. 9-13, 21-17, 22-19
File format
  Image .......................................... 20-5
File on disk
  Printing to .................................... 26-9
FileNameString
  MG ............................................... 5-7
FileVM
  Memory on disk ................................ B-2
  Virtual disk memory ........................... B-2
Fill
  Polygon .......................................... 12-5
  Small polygon .................................. 12-7
  Fill viewport ................................... 12-4
  FillArea
    MG ............................................... 12-13
FillBezier
  Fill Bézier curve ............................ 12-12
FillCircle
  MG ............................................... 12-8
  MGIW .......................................... 21-27
  MGW .......................................... 22-31
FillEllipse
  MG ............................................... 12-10
  MGIW .......................................... 21-28
  MGW .......................................... 22-32
FillEllipseSector
  MG ............................................... 12-11
  MGIW .......................................... 21-28
  MGW .......................................... 22-32
Filling
  Polygon .......................................... 21-26, 22-30
FillPolyBezier
  Fill Bézier curve ............................ 21-29, 22-33
FillPolygon
  MG ............................................... 12-5
  MGIW .......................................... 21-26
  MGW .......................................... 22-30
FillRect
  MG ............................................... 12-4
  MGIW .......................................... 21-26
  MGW .......................................... 22-30
FillSector
  MG ............................................... 12-9
  MGIW .......................................... 21-27
  MGW .......................................... 22-31
FillShortPolygon
  MG ............................................... 12-7
FillView
  MG ............................................... 12-4
FirstDisplayListPos
  MGDL .......................................... 24-21
FitIntPolyBezier
  MGBez .......................................... 31-2
FitRealPolyBezier
  MGBez .......................................... 31-3
Flicker
  Reduce .......................................... 5-24
Flip
  Image horizontally ......................... 20-17
  Image vertically .............................. 20-18
FlipImageHor
  MGImg .......................................... 20-17
G
GAClock1
Examples........................................1-11
GAClock2
Examples........................................1-13
GenericLineStyle..............................17-2
GenLineStyle
MGPat..............................................17-6
GenPattern
MGPat..............................................17-6
GetBackgroundRGB
MG................................................8-24
GetButtonPressInfo
MGMouse ............................................30-38
GetButtonReleaseInfo
MGMouse ............................................30-38
GetCharAttributeColors
Character attribute colors ..........23-27
MGIO..............................................23-4
MultiGraphics & Borland Pascal/Turbo Pascal  

DATABITEN AB

MultiGraphics  

Index - xv

GetCharAttributePos  
Character attribute position...... 23-26

GetCharAttributes  
Character attributes............... 23-24  
Get character attributes....... 23-3

GetCharData  
Get character data.................. 15-17

GetCharDataPtr  
Get character data.................. 15-19

GetCurFontInfo  
MG............................................. 15-28

GetCurSymbolsInfo  
MG............................................. 14-31

GetDeviceInfo  
MG............................................... 5-20

gGetDeviceModeInfo  
Canon Laser.............................. D14-9  
DeskJet..................................... D13-14  
Epson ........................................ D7-16  
HP LaserJet.............................. D8-10  
HPGL.......................................... D9-9  
PaintJet .................................... D12-13

GetDevicePalette  
MG............................................... 8-16

gGetDeviceSettings  
Canon Laser.............................. D14-9  
DeskJet..................................... D13-15  
Epson ........................................ D7-16  
HP LaserJet.............................. D8-11  
HPGL.......................................... D9-10  
PaintJet .................................... D12-14

GetDisplayElementPtr .......... 24-6  
MGDL........................................ 24-18

GetDisplayObjectInfo  
MGDO............................................. 25-9

GetDrawCharProc  
MGIO......................................... 23-38

GetDrawModes  
MG............................................... 9-18

GetEditModes  
MGIO......................................... 23-34

GetFastXY  
MG............................................... 9-13  
MGIW........................................ 21-17  
MGW........................................... 22-19  
Text drawing......................... 23-5

GetFirstDeviceModeInfo  
MG............................................... 5-11

GetFontInfo  
MG............................................... 15-15

GetForegroundRGB  
MG............................................... 8-22

GetHotSpot  
MG............................................... 14-29  
MGIW........................................ 21-31  
MGW........................................... 22-35

GetImageInfo............. 20-7  
MGImg........................................ 20-25

GetImagePointer  
MGMouse...................................... 30-29

GetInputProc  
MGIO......................................... 23-37

GetIOMode  
MGIO......................................... 23-22

GetMaskPointer  
MGMouse...................................... 30-22

GetMouseEventProc  
MGMouse...................................... 30-27

GetMouseMode  
MGMouse...................................... 30-24

GetMouseScrInfo  
MGMouse...................................... 30-30

GetMouseWinInfo  
MGMouse...................................... 30-37

GetNextDeviceModeInfo  
MG............................................... 5-11

GetPos  
MG............................................... 9-12  
MGIW........................................ 21-16  
MGW........................................... 22-18

GetRGB  
MG............................................... 8-18

GetRGBPalette  
MG............................................... 8-13

GetSymbolData  
Get symbol data............... 14-20

GetSymbolDataPtr  
Get symbol data............... 14-22

GetSymbolPointer  
MGMouse...................................... 30-26

GetSymbolsInfo  
MG............................................... 14-17

GetSymbolsTransform  
MG............................................... 14-32

GetTermKey  
Additional functions................. 23-7  
MGIO......................................... 23-62

GetTermKeySet  
MGIO......................................... 23-36
**Index**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetView</td>
<td>9-16</td>
</tr>
<tr>
<td>GetWinHandlesAtPos</td>
<td>7-13</td>
</tr>
<tr>
<td>GetWinMode</td>
<td>7-22</td>
</tr>
<tr>
<td>GetWorld</td>
<td>21-10</td>
</tr>
<tr>
<td>GetWorldLimits</td>
<td>21-5, 21-11</td>
</tr>
<tr>
<td>GetWorldMode</td>
<td>22-5, 22-10</td>
</tr>
<tr>
<td>GlobEP</td>
<td>3-5</td>
</tr>
<tr>
<td>GlobErrProc</td>
<td>3-5</td>
</tr>
<tr>
<td>Global error procedures</td>
<td>3-5</td>
</tr>
<tr>
<td>Global transformation</td>
<td>14-10</td>
</tr>
<tr>
<td>Graphic card Automatic detect</td>
<td>5-14</td>
</tr>
<tr>
<td>Graphics processes Concurrent</td>
<td>16-1</td>
</tr>
<tr>
<td>Gray scale Set pattern</td>
<td>17-8</td>
</tr>
<tr>
<td>Hardware palette Close</td>
<td>8-16</td>
</tr>
<tr>
<td>Hardware palette Current</td>
<td>8-16</td>
</tr>
<tr>
<td>Hardware palette Get</td>
<td>8-16</td>
</tr>
<tr>
<td>Hardware palette Set</td>
<td>8-15</td>
</tr>
<tr>
<td>HasCap</td>
<td>7-18</td>
</tr>
<tr>
<td>Heap Log</td>
<td>E-3</td>
</tr>
<tr>
<td>Heap supervising</td>
<td>E-3</td>
</tr>
<tr>
<td>Height Attribute characters</td>
<td>23-20</td>
</tr>
<tr>
<td>Height Characters</td>
<td>15-27, 21-32, 22-36</td>
</tr>
<tr>
<td>Height Convert from world coord</td>
<td>21-13, 22-14</td>
</tr>
<tr>
<td>Height Current screen</td>
<td>6-17</td>
</tr>
<tr>
<td>Height Image</td>
<td>20-27</td>
</tr>
<tr>
<td>Height Symbol</td>
<td>14-29, 21-30, 22-35</td>
</tr>
<tr>
<td>Height Window</td>
<td>7-16, 7-20</td>
</tr>
<tr>
<td>Help functions</td>
<td></td>
</tr>
<tr>
<td>Using GetTermKey</td>
<td>23-6</td>
</tr>
<tr>
<td>Hercules Colors</td>
<td>D5-5</td>
</tr>
<tr>
<td>Hercules Line style</td>
<td>D5-3</td>
</tr>
<tr>
<td>Hercules Optimizing</td>
<td>D5-2</td>
</tr>
<tr>
<td>Hercules Patterns</td>
<td>D5-3</td>
</tr>
<tr>
<td>Hercules Special screens</td>
<td>D5-2</td>
</tr>
<tr>
<td>Hercules Virtual screens</td>
<td>D5-2</td>
</tr>
<tr>
<td>Hercules Visible screen</td>
<td>D5-1</td>
</tr>
<tr>
<td>HideCursor</td>
<td>23-29</td>
</tr>
<tr>
<td>HidePointer MGI Mouse</td>
<td>30-32</td>
</tr>
<tr>
<td>HideScanWaitPointer</td>
<td>30-32</td>
</tr>
<tr>
<td>horiz</td>
<td></td>
</tr>
<tr>
<td>Hot spot Symbol</td>
<td>14-6, 14-29</td>
</tr>
<tr>
<td>HP DeskJet printers Driver</td>
<td>D13-1</td>
</tr>
<tr>
<td>HP LaserJet getDeviceModeInfo</td>
<td>D8-10</td>
</tr>
<tr>
<td>HP LaserJet getDeviceSettings</td>
<td>D8-11</td>
</tr>
<tr>
<td>HP LaserJet Line style</td>
<td>D8-3</td>
</tr>
<tr>
<td>HP LaserJet MGHPL</td>
<td>D8-1</td>
</tr>
<tr>
<td>HP LaserJet Printer fonts</td>
<td>D8-4</td>
</tr>
<tr>
<td>HP LaserJet setCharSet</td>
<td>D8-10</td>
</tr>
<tr>
<td>HP LaserJet setDensity</td>
<td>D8-9</td>
</tr>
<tr>
<td>HP LaserJet SetDeviceCall</td>
<td>D8-8</td>
</tr>
<tr>
<td>HP LaserJet The output area</td>
<td>D8-8</td>
</tr>
<tr>
<td>HP LaserJet Virtual screens</td>
<td>D8-3</td>
</tr>
</tbody>
</table>

**MultiGraphics & Borland Pascal/Turbo Pascal**

**DATABITEN AB**
<table>
<thead>
<tr>
<th>Feature</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visible screen</td>
<td>D8-2</td>
</tr>
<tr>
<td>HP PaintJet printers</td>
<td>D12-1</td>
</tr>
<tr>
<td>HPGL</td>
<td></td>
</tr>
<tr>
<td>Chord angle</td>
<td>D9-8</td>
</tr>
<tr>
<td>Colors</td>
<td>D9-5</td>
</tr>
<tr>
<td>Drawing speed</td>
<td>D9-8</td>
</tr>
<tr>
<td>Fonts</td>
<td>D9-4</td>
</tr>
<tr>
<td>getDeviceModeInfo</td>
<td>D9-9</td>
</tr>
<tr>
<td>getDeviceSettings</td>
<td>D9-10</td>
</tr>
<tr>
<td>Line style</td>
<td>D9-3</td>
</tr>
<tr>
<td>Paper sizes</td>
<td>D9-7</td>
</tr>
<tr>
<td>Patterns</td>
<td>D9-3</td>
</tr>
<tr>
<td>Pen plotter</td>
<td>D9-1</td>
</tr>
<tr>
<td>Restrictions limitations reductions</td>
<td>D9-2</td>
</tr>
<tr>
<td>Visible screen</td>
<td>D9-2</td>
</tr>
<tr>
<td>HSV</td>
<td></td>
</tr>
<tr>
<td>MGCol</td>
<td>18-1</td>
</tr>
<tr>
<td>To RGB</td>
<td>18-5</td>
</tr>
<tr>
<td>HSV technique</td>
<td>8-4</td>
</tr>
<tr>
<td>Color selection</td>
<td>8-3</td>
</tr>
<tr>
<td>HSVtoRGB</td>
<td></td>
</tr>
<tr>
<td>MGCol</td>
<td>18-5</td>
</tr>
<tr>
<td>Hue</td>
<td></td>
</tr>
<tr>
<td>Color tone</td>
<td>18-1</td>
</tr>
<tr>
<td>MGCol</td>
<td>18-1</td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>23-1</td>
</tr>
<tr>
<td>I/O errors</td>
<td>3-2</td>
</tr>
<tr>
<td>I/O mode</td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>23-23</td>
</tr>
<tr>
<td>Icons</td>
<td></td>
</tr>
<tr>
<td>Images w mask</td>
<td>20-2</td>
</tr>
<tr>
<td>IdAllocate</td>
<td>E-3</td>
</tr>
<tr>
<td>Image</td>
<td></td>
</tr>
<tr>
<td>Add mask</td>
<td>20-14</td>
</tr>
<tr>
<td>Check if monochrome</td>
<td>20-26</td>
</tr>
<tr>
<td>Check if OK</td>
<td>20-25</td>
</tr>
<tr>
<td>Clone</td>
<td>20-14</td>
</tr>
<tr>
<td>Create</td>
<td>20-8</td>
</tr>
<tr>
<td>Data</td>
<td>20-25</td>
</tr>
<tr>
<td>Deallocate</td>
<td>20-16</td>
</tr>
<tr>
<td>Draw</td>
<td>20-21</td>
</tr>
<tr>
<td>Draw part of</td>
<td>20-23</td>
</tr>
<tr>
<td>File format</td>
<td>20-5</td>
</tr>
<tr>
<td>Flip horizontally</td>
<td>20-17</td>
</tr>
<tr>
<td>Flip vertically</td>
<td>20-18</td>
</tr>
<tr>
<td>Height</td>
<td>20-27</td>
</tr>
</tbody>
</table>

**MultiGraphics & Borland Pascal/Turbo Pascal**

**DATABITEN AB**
InitCOM2
  MGOut......................................... 26-8
InitDISK
  MGOut......................................... 26-10
Input
  Text based................................... 23-6
Input devices
  Polling....................................... 1-8
Input operations
  MGIO......................................... 23-12
Input procedure
  Empty ........................................ 23-10
  MGIO......................................... 23-13
  Read which ................................ 23-37
  Set ........................................... 23-36
Input procedures
  User defined............................... 23-7
InputData
  MGIO......................................... 23-12
InputProc
  MGIO......................................... 23-13
InputType
  MGIO......................................... 23-12
InsertDisplayElement
  Insert element .............................. 24-6
  MGDL........................................ 24-18
InsertFromDisplayFile
  MGDL........................................ 24-25
InstallEventServer
  MGEv ......................................... 27-10
InstallGlbEP
  Install error procedures.................. 3-6
IntCheck
  MGIO......................................... 23-54
Integer
  Control procedure.................................. 23-54
  Read........................................... 23-57, 23-58
  Read long..................................... 23-59
  Write........................................... 23-40, 23-41
  Write long..................................... 23-42
Intensity..................................... 18-5, 18-6
IntWorldMode
  MGIW........................................ 21-6
invalidEventGenerator
  MGEv ......................................... 27-7
invalidHSV................................... 18-2
invalidServerProc
  MGEv ......................................... 27-7
IO mode
Key Pressed Event
  MG Kb ........................................ 29-7
KeyPressedProc
  MG ................................................. 4-2
keys
  MGIO ........................................... 23-9
Termination........................................ 23-35
  Toggle.......................................... 29-4
KeyPressedEvent
  Wait on ........................................ 23-45

L
Landscape
  Orientation ..................................... 5-8
leftButtonDoubleClick
  MG Mouse ..................................... 30-11
leftButtonPressed
  MG Mouse ..................................... 30-11
leftButtonReleased
  MG Mouse ..................................... 30-11
Limiting window
  Mouse .......................................... 30-34
Limits
  World coordinates ..................... 21-11, 22-12
Line
  Clear to end of line ...................... 23-21
  Draw distance .......................... 11-10, 21-20, 22-23
  Draw relative ............................. 11-9, 21-20, 22-23
  Draw to ..................................... 11-9, 21-20, 22-22
  New ............................................. 23-38
  Skip endpoint............................... 11-7
Line style
  8514/A ........................................ D6-4
  Ask if protected ......................... 11-7
  Canon Laser ................................ D14-3
  CGA ........................................... D1-3
  Current position .......................... 11-6
  Data types .................................. 11-4
  DeskJet ..................................... D13-4
  EGA ........................................... D3-4
  Epson ......................................... D7-4
  Generic ...................................... 17-6
  Hercules .................................... D5-3
  HP LaserJet .................................. D8-3
  HPGL .......................................... D9-3
  MCGA .......................................... D2-2
  PaintJet ..................................... D12-4
  Protect ....................................... 11-6
  Read current ................................ 11-5
  Set ............................................. 11-4
  Set generic .................................. 17-6
  Set position in ........................... 11-5
  SVG A ........................................ D1-3
  Toshiba ....................................... D10-2
  VGA ........................................... D4-4
Line type
  Ask if protected ......................... 11-7
  Generic ...................................... 17-6
  Line Style .................................... 11-4
  Protect ....................................... 11-6
  Set ............................................. 11-4
  Set generic .................................. 17-6
Line width
  Read current ................................ 11-8
  Set ............................................. 11-8
Lines
  Summary ..................................... 11-1
Line Style
  Data types .................................. 11-4
  MGPat .......................................... 17-1
Link
  MG .............................................. 24-4, 24-13
  MGI Img .................................... 24-4, 24-13
  MGIO ........................................... 24-4, 24-13
  MG IW ........................................ 24-4, 24-13
  MGPat ........................................ 24-4, 24-13
  MG W .......................................... 24-4, 24-13
  MG .............................................. 24-4, 24-13
  MG .............................................. 24-4, 24-13
Link procedures
  Summary .................................... 24-13
Load
  Driver ........................................... 5-3
  Load automatic
    Driver ........................................... 5-14
  Load image ................................... 20-12
  LoadBGIFont
    MG Fnt ....................................... 19-2
  LoadBGISymbols
    MG Fnt ....................................... 19-3
  LoadDevice
    MG .............................................. 5-12
  LoadDisplayObjects
    MG DO ........................................ 25-14
  LoadFont
    MG .............................................. 15-12
  LoadImage
    MG Img ....................................... 20-12
  LoadNamedDisplayObjects
    MG DO ........................................ 25-15
  LoadPCXImage
    MG Img ....................................... 20-13
xx - Index

LoadSymbols
MG............................................. 14-15

Log
Events ............................................E-6
Memory ............................................E-3

logAllKbdEvents
MGKbd........................................ 29-3

logKbdKey
MGKbd........................................ 29-3

LongInt
Read........................................... 23-59
Write.......................................... 23-42

LPT1
MGOut......................................... 26-7

LPT2
MGOut......................................... 26-7

LPT3
MGOut......................................... 26-7

M

Mask
Drawing mode ............................... 9-9
Image ........................................... 20-3, 20-27
Images........................................... 20-1
Link to image ................................. 20-14

Mask screen
Image ........................................... 20-7

Masked mouse pointer .................... 30-15

MaskPointer
MGMouse ........................................ 30-15

matchAllEvents
MGEv .......................................... 27-9

matchAnyEvent
MGEv .......................................... 27-9

MatchBackground
MG............................................... 8-23

matchEventsExactly
MGEv .......................................... 27-9

MatchForeground
MG............................................... 8-22

MatchingBackground
MG............................................... 8-20

MatchingForeground
MG............................................... 8-20

maxDisplayObjects
MGDO ........................................... 25-6

maxEventGen
MGEv .......................................... 27-7

maxHue ......................................... 18-2

maxRamSymbolsSize
MG ............................................. 14-13

MCGA
Colors.......................................... D2-4
Line style ..................................... D2-2
Patterns........................................ D2-2
Virtual screens .............................. D2-1
Visible screen ............................... D2-1

Memory
EmsVM ........................................... B-2
Images........................................... 20-4
Virtual ........................................... B-1
XMS ............................................. B-2

MG
Introduction .................................. 2-1

MGBGI
BGI emulation .................................. D-1

MGCanon
Canon Laser.................................. D14-1
Orientations.................................. D14-8

MGCGA
CGA............................................. D1-1

MGClock
Clock events ................................. 28-1

MGCol
HSV color model ............................. 18-1
HSV technique ............................... 8-4

MGDebug
Debugging...................................... E-1

MGDJet
Printers ........................................ D13-1

MGDL87
versus MGDL.................................. 24-10

MGDO
Display objects ............................. 25-1

MGDP............................................... 24-6

MGEGA
EGA ............................................. D3-1

mgEps
Printers ........................................ D7-1

MGErr
Error codes ................................... 3-2

MGEv
Event handling .............................. 27-1

MGFnt - Font conversion .................. 19-1

MGHPGL.......................................... D9-1

MGHPL
HP LaserJet .................................... D8-1
Orientations.................................. D8-9
<table>
<thead>
<tr>
<th>MultiGraphics &amp; Borland Pascal/Turbo Pascal</th>
<th>DATABITEN AB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGImg Images</td>
<td>20-1</td>
</tr>
<tr>
<td>MGIO Summary</td>
<td>23-1</td>
</tr>
<tr>
<td>MGIO.cap</td>
<td>23-15</td>
</tr>
<tr>
<td>MGIO</td>
<td>23-1</td>
</tr>
<tr>
<td>MGIW Versus MGW</td>
<td>21-2</td>
</tr>
<tr>
<td>World coordinates</td>
<td>21-1</td>
</tr>
<tr>
<td>MGIW</td>
<td>21-1</td>
</tr>
<tr>
<td>MGMouse Mouse handling</td>
<td>30-1</td>
</tr>
<tr>
<td>MGOut Printing</td>
<td>26-1</td>
</tr>
<tr>
<td>MGPat Error handling</td>
<td>17-1</td>
</tr>
<tr>
<td>Patterns</td>
<td>17-1</td>
</tr>
<tr>
<td>MGPat.SetGenLineStyle Line style</td>
<td>11-5</td>
</tr>
<tr>
<td>MGJet Printers</td>
<td>D12-1</td>
</tr>
<tr>
<td>MGVGA VGA</td>
<td>D4-1</td>
</tr>
<tr>
<td>MGW Versus MGIW</td>
<td>22-2</td>
</tr>
<tr>
<td>World coordinates</td>
<td>22-1</td>
</tr>
<tr>
<td>middleButtonDoubleClick MGMouse</td>
<td>30-11</td>
</tr>
<tr>
<td>middleButtonPressed MGMouse</td>
<td>30-11</td>
</tr>
<tr>
<td>middleButtonReleased MGMouse</td>
<td>30-11</td>
</tr>
<tr>
<td>minHue</td>
<td>18-2</td>
</tr>
<tr>
<td>Mirror Image horizontally</td>
<td>20-17</td>
</tr>
<tr>
<td>Image vertically</td>
<td>20-18</td>
</tr>
<tr>
<td>Mix Drawing mode</td>
<td>9-8</td>
</tr>
<tr>
<td>Mode Device</td>
<td>5-16</td>
</tr>
<tr>
<td>Window</td>
<td>7-21</td>
</tr>
<tr>
<td>Mode number Device</td>
<td>5-22</td>
</tr>
<tr>
<td>Modes Devices</td>
<td>5-2</td>
</tr>
<tr>
<td>Mono virtual screen</td>
<td>6-5</td>
</tr>
<tr>
<td>Create</td>
<td>6-9</td>
</tr>
<tr>
<td>Monochrome image Check if</td>
<td>20-26</td>
</tr>
<tr>
<td>Read</td>
<td>20-11</td>
</tr>
<tr>
<td>Monochrome screen</td>
<td>6-6</td>
</tr>
<tr>
<td>monoVirtScr Screen type</td>
<td>6-6</td>
</tr>
<tr>
<td>Mouse exist</td>
<td>30-18</td>
</tr>
<tr>
<td>hardware requirements</td>
<td>30-2</td>
</tr>
<tr>
<td>Limit to window</td>
<td>30-34</td>
</tr>
<tr>
<td>The number of buttons</td>
<td>30-18</td>
</tr>
<tr>
<td>User mouse event procedure</td>
<td>30-23</td>
</tr>
<tr>
<td>Within window?</td>
<td>30-37</td>
</tr>
<tr>
<td>Mouse acceleration</td>
<td>30-35</td>
</tr>
<tr>
<td>Mouse buttons Read release of</td>
<td>30-38</td>
</tr>
<tr>
<td>Read the pressing of</td>
<td>30-38</td>
</tr>
<tr>
<td>Use of</td>
<td>30-3</td>
</tr>
<tr>
<td>Mouse emulation Only</td>
<td>30-39</td>
</tr>
<tr>
<td>Mouse event Emulate</td>
<td>30-40</td>
</tr>
<tr>
<td>Mouse event procedure Read current</td>
<td>30-24</td>
</tr>
<tr>
<td>User written</td>
<td>30-14</td>
</tr>
<tr>
<td>Mouse event procedures User</td>
<td>30-8</td>
</tr>
<tr>
<td>Mouse event type User</td>
<td>30-8</td>
</tr>
<tr>
<td>Mouse event types Summary</td>
<td>30-3</td>
</tr>
<tr>
<td>Mouse events Event procedures</td>
<td>30-8</td>
</tr>
<tr>
<td>Start</td>
<td>30-18</td>
</tr>
<tr>
<td>Stop</td>
<td>30-19</td>
</tr>
<tr>
<td>Mouse graphics device Current</td>
<td>30-22</td>
</tr>
<tr>
<td>Mouse information Read screen data</td>
<td>30-37</td>
</tr>
<tr>
<td>Read window data</td>
<td>30-37</td>
</tr>
<tr>
<td>Mouse mode MGMouse</td>
<td>30-16</td>
</tr>
<tr>
<td>Read</td>
<td>30-20</td>
</tr>
<tr>
<td>Save</td>
<td>30-8</td>
</tr>
<tr>
<td>Set</td>
<td>30-21</td>
</tr>
<tr>
<td>Mouse pointer Background color</td>
<td>30-31</td>
</tr>
<tr>
<td>Drawing modes</td>
<td>30-6</td>
</tr>
<tr>
<td>Foreground color</td>
<td>30-30</td>
</tr>
<tr>
<td>Hide</td>
<td>30-32</td>
</tr>
<tr>
<td>Image type</td>
<td>30-16, 30-28, 30-29</td>
</tr>
<tr>
<td>Masked</td>
<td>30-27, 30-28</td>
</tr>
<tr>
<td>Read</td>
<td>30-26</td>
</tr>
</tbody>
</table>

**MultiGraphics Index - xxi**

**MultiGraphics & Borland Pascal/Turbo Pascal**

**DATABITEN AB**
Reduced flicker .......................... 30-33
Show ...................................... 30-31
Specify ................................... 30-25
Symbol type ............................. 30-15
Visibility ................................. 30-7, 30-32
With mask ................................ 30-15
Mouse pointer type
MGMouse .................................. 30-15
Read ....................................... 30-30
Mouse pointer types
Summary .................................. 30-3
Mouse position
Set ......................................... 30-36
Window coordinates .................. 30-36
Mouse protection
Set way ..................................... 30-34
Mouse screen
Current ................................... 30-22
Mouse sensitivity ...................... 30-35
Mouse window
Current ................................... 30-22
MOUSE.COM .............................. 30-2
MOUSE.SYS ............................... 30-2
MouseButtons
MGMouse .................................. 30-18
MouseEvent
MGMouse .................................. 30-14
MouseEventProc ......................... 30-8
 MGMouse .............................. 30-8, 30-14
MouseExist
MGMouse .................................. 30-18
mouseGen
MGEv ....................................... 27-7
MouseInWin
MGMouse .................................. 30-37
MouseMode
MGMouse .................................. 30-16
mouseMoved
MGMouse .................................. 30-11
MousePointerType
MGMouse .................................. 30-15
MovePos
MG ......................................... 9-12
MGIW ...................................... 21-16
MGW ....................................... 22-17
msPerTick
MG .......................................... 2-3
Multi-tasking ............................. 16-1
MultiGraphics
Let's Draw! ................................. 1-1
Multiple passes
DeskJet .................................... D13-12
N
New line
Write ...................................... 23-38
NextDisplayListPos
MGDL ...................................... 24-20
NextEventGen
MGEv ....................................... 27-7
Normalized world coordinates
21-4, 22-4
nullDisplayList
Empty display list .................... 24-11
nullDisplayListPos
MGDL ...................................... 24-14
Non-existent display list position
24-14
nullDisplayObjects
MGDO ...................................... 25-6
NullDrawChar
MGIO ....................................... 23-10
nullFont
MG .......................................... 15-11
nullGen
MGEv ....................................... 27-7
nullImage
MGImg ..................................... 20-6
NullInput
MGIO ....................................... 23-10
nullLineStyle
MG .......................................... 11-4
NullMouseEvent
MGMouse .................................. 30-13
nullPattern
MG .......................................... 12-2
nullPoint ................................. 10-2, 22-7
MG .......................................... 11-11
MGIW ...................................... 21-21, 21-26
MGW ....................................... 22-24, 22-25, 22-31
nullScreen
MG .......................................... 6-7
nullShortPoint ........................... 10-2
NullStrConversion
MGIO ....................................... 23-10
nullSymbols
MG .......................................... 14-13
nullWindow
MG .......................................... 7-7
MultiGraphics

O

off
  Constant......................................... 2-2
ok
  SV .................................................. 3-6
on
  Constant......................................... 2-2
OnOutlineProc
  MG............................................... 10-2
Optimization
  Speed SuperVGA................................. D11-4
Optimizing
  8514/A ........................................ D6-4
  CGA ............................................. D1-2
  EGA ............................................. D3-3
  Hercules ....................................... D5-2
  Toshiba ....................................... D10-2
  VGA ............................................. D4-4
OR
  Drawing mode .................................. 9-8
Orientation ......................................... 5-7
  Canon Laser .................................... D14-8
  Device .......................................... 5-22
  HP Laser ........................................ D8-9
  Landscape ....................................... 5-8
  Portrait ........................................... 5-8
  Set ................................................ 5-17
OrientationSet .................................... 5-7
Outline font ....................................... 15-2, 15-4
Outline symbols .................................. 14-1, 14-3
Outline tracing
  TraceComplexOutline ............................ 10-6
  TraceOutline ..................................... 10-5
OutProc
  Example ......................................... 26-2
  Output procedures .............................. 26-5
Output
  User-defined procedures ...................... 23-7
Output area
  Canon Laser ..................................... D14-7
  DeskJet ......................................... D13-10
  Epson ............................................ D7-10
  HP LaserJet ..................................... D8-8
  PaintJet ........................................ D12-11
Output routine
  Device .......................................... 5-18

P

PaintJet

Index - xxiii

Color graphics............................ D12-7
Colors ........................................ D12-3, D12-9
Compression ............................... D12-12
getDeviceModeInfo ..................... D12-13
getDeviceSettings ...................... D12-14
Line style ...................................... D12-4
Pattern ........................................ D12-4
Printer fonts ............................... D12-5
setDensity ..................................... D12-11
setSize ......................................... D12-11
Virtual screens ............................. D12-3
Visible screen ................................ D12-2
PaintJet printers
  Driver .......................................... D12-1
Palette
  8514/A ........................................ D6-7
  CGA ............................................. D1-5
  Close hardware ............................... 8-16
  Close variable ............................... 8-14
  Create RGB ................................... 8-14
  Current ........................................ 8-5
  Current hardware ............................ 8-16
  DeskJet ........................................ D13-9
  EGA ............................................. D3-6
  Epson .......................................... D7-9
  Get hardware ................................. 8-16
  Hercules ....................................... D5-5
  HPGL ............................................ D9-5
  Image .......................................... 20-4, 20-28
  MCGA .......................................... D2-4
  PaintJet ........................................ D12-9
  Read RGB ...................................... 8-13
  Set default ................................... 8-11
  Set hardware .................................. 8-15
  Set RGB ....................................... 8-12
  Summary ....................................... 8-4
  SVGA .......................................... D11-8
  Toshiba ....................................... D10-4
  VGA ............................................. D4-7
Palette colors
  Set RGB ........................................ 8-17
Paper sizes
  HPGL ............................................ D9-7
Parity
  MGOOut ......................................... 26-4
  Printing ........................................ 26-4
Pattern
  8514/A ........................................ D6-5
  Data type ....................................... 12-2
  DeskJet ........................................ D13-4
  Epson ......................................... D7-4, D8-4, D14-3

MultiGraphics & Borland Pascal/Turbo Pascal  DATABITEN AB
Fill circle ..................................... 12-8
Fill ellipse ........... 12-10, 21-28, 22-32
Fill ellipse sector ........ 12-11
Fill rectangle......... 12-4, 21-26, 22-30
Fill sector of ellipse......21-28, 22-32
Fill viewport ................................ 12-4
Fill with ..............21-26, 22-30
Filling circles........21-27, 22-31
MGIW........................................ 21-26
MGPat.......................................... 17-1
MGW ......................................... 22-30
PaintJet ..................................... D12-4
Read the current ..........12-3
Set generic ..............17-7
Set the current.................. 12-2
Summary.............................. 12-1
Patterns
bitPat8.......................................... 17-4
CGA............................................ D1-3
EGA........................................... D3-4
Generic................................. 17-7
Hercules................................. D5-3
HPGL.......................................... D9-3
MCGA........................................ D2-2
SVGA........................................ D11-4
Toshiba................................. D10-3
VGA........................................... D4-4
Pause
MG................................................. 4-3
PC Paintbrush.................. 20-1
PCC image ............................... 20-5
Load........................................... 20-13
PCX files
DeskJet................................. D13-5
Epson............................... D7-5
PaintJet............................... D12-5
PCX format
8514/A............................... D6-6
Canon Laser........................... D14-5
CGA........................................... D1-4
EGA........................................... D3-5
Hercules................................. D5-4
HP LaserJet........................... D8-6
Images............................... 20-1, 20-5
MCGA............................... D2-3
SVGA............................... D11-6
Toshiba................................. D10-3
VGA........................................... D4-5
PCX image
Load........................................... 20-13
Save........................................ 20-15
PC Paintbrush ................................ 20-1
Peek
SetPeekProc.............................. 26-5
Pieslice
Fill........................................ 12-9, 21-27, 22-31
Pixel color
Read........................................ 10-4
Place
Position.................................. 9-1
Playback
Display list ................................ 24-4
Plotter
HPGL........................................ D9-1
Point
Draw........................................ 10-3
MG............................................. 10-2
MGW........................................... 22-8
Short type................................ 10-2
Point color
Read........................................ 10-4
Point drawing
MG............................................. 10-1
PointDeviceColor
MG............................................. 10-4
MGIW........................................ 21-19
MGW........................................... 22-22
Pointer
Hide....................................... 30-32
Hide mouse.......................... 30-32
Mouse30-3, 30-25, 30-26, 30-27, 30-28, 30-29
Show mouse.......................... 30-31
PointerVisible.......................... 30-32
Polling
Input devices.................. 1-8
Poly Bézier
Transform.............................. 22-29
Polybézier
Draw.................. 11-23, 21-25, 22-29
Fill........................................ 12-12
Filling of ................. 21-29, 22-33
Several Bézier curves ........11-3
Transform.............................. 11-22
Polygon
Pattern fill ....12-5, 12-7, 21-26, 22-30
Polyline
Draw........................................ 11-12
Transform.................. 11-11, 11-14, 22-24
MultiGraphics

Index - xxv

Polylines
  Draw ................... 11-12, 21-21, 22-24
  Draw short .................. 11-14

Portability
  Display list................... 24-9
  Display objects ............... 25-4

Portrait
  Orientation ................ 5-8

Position
  Current window .................. 7-19
  Fast ................ 9-13, 21-17, 22-19
  Get current ...................... 9-12
  Move distance .... 9-12, 21-16, 22-17
  Read ................ 21-16, 22-18
  Set ....... 21-15, 22-16
  Set current ................. 9-11
  Set relative .... 9-11, 21-15, 22-17
  Summary ................... 9-1
  Text column .......... 23-17
  Window .............. 7-15, 7-20

PrevDisplayListPos
  MGDL .................................... 24-21

Primary colors
  RGB ........................................ 8-2

Printer
  Flush screen ..................... 6-15

Printers
  MGDJet .................. D13-1
  MGEps ................ D7-1
  MGPJet ................ D12-1
  Parallel .................... 26-7

Printing
  Peeking ....................... 26-5

Printing routine
  Device ....................... 5-18

Printout procedures
  Enumeration of ............... 26-4

Processes
  Concurrent ..................... 16-1

Proportional font ................. 15-6

Proposal mode
  Is active? ....................... 23-33
  Set ................... 23-33

R

Raster
  Density ................... 17-3
  Set ....................... 17-8
  rasterDevice ................ 5-7
  rasterVectorDevice ........ 5-7
  RdChar
    MGIO ................................ 23-55
  RdEventKey
    MGKbd ................................ 29-7
  RdFixReal
    MGIO ................................ 23-61
  RdInt
    MGIO ................................ 23-58
  RdKbd
    MGIO ................................ 23-45
  RdKey
    MG ........................................ 4-2
  RdKeyProc
    MG ........................................ 4-2
  RdLong
    MGIO ................................ 23-59
  RdMatchingEvent
    MGEv ................................ 27-16
  RdNextEvent
    MGEv ................................ 27-15
  RdReal
    MGIO ................................ 23-60
  Rdst
    MGIO ................................ 23-56
  RdStr
    MGIO ................................ 23-57

Read
  Bit block ......................... 13-9
  Characters ................... 23-55
  FixReal ................... 23-61
  In graphics mode .............. 23-5
  INTEGER ...................... 23-58
  LongInt ..................... 23-59
  Real ....................... 23-60
  String ...................... 23-56
  Termination key .............. 23-62
  Text ..................... 23-6
  Word ..................... 23-57

ReadBitBlock
  MG ........................................ 13-9

ReadImage
  MGImg ......................... 20-9

ReadImageNoPal
  MGImg ......................... 20-11

ReadLn
  In graphics mode .............. 23-5

ReadMonoImage
  MGImg ......................... 20-11

Real
  MG ........................................ 2-4
Index

Read ........................................... 23-60
Read fixed.................................. 23-61
Write fix ................................... 23-43
RealCheck
MGIO......................................... 23-55
Recording
Display list................................. 24-3
Start begin.................................. 24-12
Stop............................................ 24-12
Rectangle
Draw ........................................ 11-15, 11-16
Pattern fill................................. 12-4, 21-26, 22-30
Rectangle area
Copy ....................................... 13-4, 21-29, 22-34
Rectangles
Draw ..................................... 21-21, 21-22, 22-25
Reduce
Screen .................................... 6-11, 13-7
Reference
Display lists ................................ 24-8
Relative position
Set .................................. 9-11, 21-15, 22-17
RemoveEventServer
MGEv ....................................... 27-11
Replace
Drawing mode ............................. 9-7
ResetLoader
Choice of loading technique ......... 24-9
MGDL........................................ 24-23
ResetOneTimeAlarm
MGClock ................................ 28-13
Result
SV ........................................... 3-7
Result codes ............................. 3-2
RGB ....................................... 18-5, 18-6
To HSV ................................... 18-6
RGB color
Get colors................................ 8-18
Set palette colors...................... 8-17
RGB palette ..................... 20-7
Close variable ..................... 8-14
Create variable .................. 8-14
Image ................................ 20-28
Read .................................. 8-13
Set ...................................... 8-12
RGB value ..................... 8-2
RGBtoHSV
MGCol ................................... 18-6
rightButtonDoubleClick
MGMouse .................................. 30-11
rightButtonText
MGMouse .................................. 30-11
rightButtonReleased
MGMouse .................................. 30-11
Rotate
Fonts ................................... 15-9
Image 8- degrees ........... 20-18
Image 90 degrees ........ 20-18, 20-19
Symbols ....................... 14-10, 14-31
RotCopyImage90
MGImg .................................. 20-19
RotImage180
MGImg .................................. 20-18
RotImage90
MGImg .................................. 20-18
Row
Read current ..................... 23-18
Row,column
Set .................................. 23-17
Rows
Number of in window .......... 23-16
RS232
Initialize ............... 26-8

S
Saturation
Colors .................................. 18-1
MGCol .................................. 18-1
Save image .......................... 20-15
SaveDisplayObjects
MGDO..................................... 25-14
SaveFont
MG ................................... 15-13
SaveImage
MGImg .................................. 20-15
SavePCXImage
MGImg .................................. 20-15
SaveSymbols
MG ................................... 14-16
SBL
Bitmapped large symbols ......... 14-8
SBS
Bitmapped small symbols ....... 14-8
Scale
Fonts .................................. 15-9
Symbols ..................... 14-10, 14-31
Scan code
KbdCode ................................ 4-2
ScanWait
MG .................................................. 5-24
Screen
Activate ........................................ 6-14
Clone .............................................. 6-10
Concepts ......................................... 6-1
Create mono virtual ....................... 6-9
Create special .............................. 6-10
Create virtual ............................... 6-7
Current .................................. 6-3, 6-14
Dispose ........................................ 6-13
Flush visible .................................. 6-15
For mouse .................................. 30-22
Increase capability....................... 6-13
Maximum color mask .................... 6-17
MG............................................... 6-6
Set visible .................................... 6-15
Virtual in image ......................... 20-26
Zoom........................................ 6-11, 13-7
Screen capability
Increase ........................................ 6-6
Screen height
Current ......................................... 6-17
Screen type ........................................ 6-6
Current ......................................... 6-16
Screen width
Current ......................................... 6-16
Screen window ................................ 7-2
Check ........................................ 7-18
ScreenType
MG............................................... 6-6
ScreenTypeSet
Screen types ........................................ 6-6
Scroll
MGIO ........................................... 23-44
Read ........................................... 23-23
Window ........................................ 23-23
Scroll roll
Window ........................................ 23-44
Scrolling
Summary ......................................... 23-4
Sector
Draw .................. 11-19, 21-23, 22-27
Fill .................. 12-9, 21-27, 22-31
Sector of ellipse
Pattern fill .......... 21-28, 22-32
Sectors of ellipses
Draw .................. 11-21, 21-25, 22-28
Sensitivity
Mouse ........................................ 30-35
Serial communication .............. 26-8
Serial port
Initialize ........................................ 26-8
Service routines
MG ............................................... 27-5
SetAspect
MG ............................................... 5-23
SetBackgroundColor
MG ............................................... 8-23
SetBeep
MGIO ........................................... 23-28
SetBlackOnWhite
MG ............................................... 23-32
SetColorMask
MG ............................................... 8-19
SetCharAttributeColors
Character attribute colors ........ 23-26
MGIO ........................................... 8-6
SetCharAttributePos
Character attribute position 23-25
SetCharAttributes
Character attributes .... 23-24
Set character attributes ... 23-3
setCharSet
HP LaserJet ......... D8-10
SetColorMask
MG ............................................... 9-16
SetColRow
MGIO ........................................... 23-17
SetCursor
MGIO ........................................... 23-28
SetCursorBlink
MGIO ........................................... 23-31
SetCursorColor
MGIO ........................................... 23-30
setDefaultFont
MG ............................................... 15-24
setDefaultPalette
MG ............................................... 8-11
setDefaultSymbols
MG ............................................... 14-27
setDensity
Canon Laser .............. D14-8
DeskJet .............. D13-11
Epson .............. D7-10
HP LaserJet .............. D8-9
PaintJet .............. D12-11
SetDevice
MG ............................................... 5-21
Visible screen .......... 6-3
SetDeviceCall
Canon Laser.............................. D14-7
HP LaserJet................................. D8-8
MG............................................... 5-18

SetDeviceMode
MG............................................... 5-16

SetDeviceOutput
MG............................................... 5-7
MG............................................... 5-18

SetDevicePalette
MG............................................... 8-15

SetDir
MG............................................... 9-14
MGW ............................................ 22-19

SetDisplayListPos
MGDL........................................ 24-20

SetDoubleClickInterval
MGMouse ...................................... 30-35

SetDoubleSpeedThreshold
MGMouse ...................................... 30-35

SetDrawCharProc
MGIO......................................... 23-37

SetDrawMode
MG............................................... 9-17

SetEditModes
MGIO......................................... 23-34

SetEmulationOnly
MGMouse ...................................... 30-39

SetEndPointSkip
MG............................................... 11-7

SetEP .................................................. 3-3

SetErrorWin
MG............................................... 3-4

SetFirstDisplayListPos
MGDL........................................ 24-21

SetFont
MG............................................... 15-24

SetForeground
MG............................................... 8-21

SetGenLineStyle
MGPat.......................................... 17-6

SetGenPattern
MGPat.......................................... 17-7

SetHalftone
MGPat.......................................... 17-8

SetImagePointer
MGMouse ...................................... 30-28

SetInputProc
MGIO......................................... 23-36

SetIOMode
MGIO......................................... 23-23

SetLineStyle
MG............................................... 11-4

SetLineStylePos
MG............................................... 11-5

SetLineWidth
MG............................................... 11-8

SetMaskPointer
MGMouse ...................................... 30-27

SetMouseEventProc
MGMouse ...................................... 30-23

SetMouseMode
MGMouse ...................................... 30-21

SetMouseScrPos
MGMouse ...................................... 30-36

SetMouseSens
MGMouse ...................................... 30-35

SetMouseWinLimits
MGMouse ...................................... 30-34

SetMouseWinPos
MGMouse ...................................... 30-36

SetMSWindowsPalette
Set Windows colors ..................... 18-6

SetOneTimeAlarm
MGClock.................................... 28-12

SetOrientation
MG............................................... 5-17

SetPattern.......................................... 17-4

MG............................................... 12-2

SetPeekProc
MGOut ......................................... 26-5

SetPointerBG
MGMouse ...................................... 30-31

SetPointerFG
MGMouse ...................................... 30-30

SetPointerProtection
MGMouse ...................................... 30-34

SetPointerRedrawTime ..................30-33

SetPos
MG............................................... 9-11
MGW ............................................ 22-16

SetProposal
MGIO......................................... 23-33

SetProtectLineStyle
MG............................................... 11-6

SetReducedFlicker
MGMouse ...................................... 30-33
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetRelPos</td>
<td>9-11</td>
</tr>
<tr>
<td>MGIW</td>
<td>21-15</td>
</tr>
<tr>
<td>MGW</td>
<td>22-17</td>
</tr>
<tr>
<td>SetRepeatedAlarm</td>
<td>28-13</td>
</tr>
<tr>
<td>MGclock</td>
<td>24-23</td>
</tr>
<tr>
<td>SetRGB</td>
<td>8-17</td>
</tr>
<tr>
<td>MGion</td>
<td>11-15</td>
</tr>
<tr>
<td>MGIW</td>
<td>21-12</td>
</tr>
<tr>
<td>MGW</td>
<td>22-20</td>
</tr>
<tr>
<td>SetRGBPalette</td>
<td>8-12</td>
</tr>
<tr>
<td>SetScr</td>
<td>6-14</td>
</tr>
<tr>
<td>MG</td>
<td>7-17</td>
</tr>
<tr>
<td>SetScrollMode</td>
<td>23-23</td>
</tr>
<tr>
<td>MG</td>
<td>7-17</td>
</tr>
<tr>
<td>SetScrWin</td>
<td>5-16</td>
</tr>
<tr>
<td>MG</td>
<td>7-17</td>
</tr>
<tr>
<td>setSize</td>
<td></td>
</tr>
<tr>
<td>Canon Laser</td>
<td>D14-7</td>
</tr>
<tr>
<td>DeskJet</td>
<td>D13-10</td>
</tr>
<tr>
<td>Epson</td>
<td>D7-10</td>
</tr>
<tr>
<td>HP LaserJet</td>
<td>D8-8</td>
</tr>
<tr>
<td>HPGL plotters</td>
<td>D9-7</td>
</tr>
<tr>
<td>PaintJet</td>
<td>D12-11</td>
</tr>
<tr>
<td>setSmoothness</td>
<td></td>
</tr>
<tr>
<td>HPGL-plotters</td>
<td>D9-8</td>
</tr>
<tr>
<td>setSpeed</td>
<td></td>
</tr>
<tr>
<td>HPGL plotters</td>
<td>D9-8</td>
</tr>
<tr>
<td>SetSymbolPointer</td>
<td>30-25</td>
</tr>
<tr>
<td>MGmouse</td>
<td></td>
</tr>
<tr>
<td>SetSymbols</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>14-27</td>
</tr>
<tr>
<td>SetSymbolsTransform</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>14-31</td>
</tr>
<tr>
<td>SetTermKeySet</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>23-35</td>
</tr>
<tr>
<td>setTypeFace</td>
<td></td>
</tr>
<tr>
<td>Epson</td>
<td>D7-11</td>
</tr>
<tr>
<td>setTypeFaceSize</td>
<td></td>
</tr>
<tr>
<td>Epson</td>
<td>D7-13</td>
</tr>
<tr>
<td>setTypeFaceStyle</td>
<td></td>
</tr>
<tr>
<td>Epson</td>
<td>D7-12</td>
</tr>
<tr>
<td>setTypeFaceWeight</td>
<td></td>
</tr>
<tr>
<td>Epson</td>
<td>D7-12</td>
</tr>
<tr>
<td>SetUnderline</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>23-27</td>
</tr>
<tr>
<td>SetView</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>9-15</td>
</tr>
<tr>
<td>MGIW</td>
<td>21-18</td>
</tr>
<tr>
<td>MGW</td>
<td>22-20</td>
</tr>
<tr>
<td>SetVisiScr</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>6-15</td>
</tr>
<tr>
<td>SetVM</td>
<td></td>
</tr>
<tr>
<td>VM</td>
<td>B-5</td>
</tr>
<tr>
<td>SetWin</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>7-16</td>
</tr>
<tr>
<td>SetWinMode</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>7-21</td>
</tr>
<tr>
<td>SetWorld</td>
<td></td>
</tr>
<tr>
<td>MGIW</td>
<td>21-9</td>
</tr>
<tr>
<td>MGW</td>
<td>22-11</td>
</tr>
<tr>
<td>Shadows</td>
<td></td>
</tr>
<tr>
<td>MGIW</td>
<td>23-3</td>
</tr>
<tr>
<td>Shear</td>
<td></td>
</tr>
<tr>
<td>Fonts</td>
<td></td>
</tr>
<tr>
<td>Symbols</td>
<td>14-10, 14-31</td>
</tr>
<tr>
<td>ShiftKey</td>
<td></td>
</tr>
<tr>
<td>MGKbd</td>
<td></td>
</tr>
<tr>
<td>ShiftKeySet</td>
<td></td>
</tr>
<tr>
<td>MGKbd</td>
<td></td>
</tr>
<tr>
<td>Shingling</td>
<td></td>
</tr>
<tr>
<td>DeskJet</td>
<td>D13-12</td>
</tr>
<tr>
<td>ShinglingSet</td>
<td></td>
</tr>
<tr>
<td>MGDJet</td>
<td>D13-13</td>
</tr>
<tr>
<td>shortNullPoint</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>11-14</td>
</tr>
<tr>
<td>ShortPoint</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td></td>
</tr>
<tr>
<td>ShowCursor</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td>23-29</td>
</tr>
<tr>
<td>ShowPointer</td>
<td></td>
</tr>
<tr>
<td>MGIW</td>
<td>30-31</td>
</tr>
<tr>
<td>SOL</td>
<td></td>
</tr>
<tr>
<td>Large outline symbols</td>
<td>14-8</td>
</tr>
<tr>
<td>Small outline symbols</td>
<td>14-8</td>
</tr>
<tr>
<td>solid</td>
<td>17-2</td>
</tr>
<tr>
<td>solidPat</td>
<td>17-2</td>
</tr>
<tr>
<td>sparse8</td>
<td>17-4</td>
</tr>
<tr>
<td>Special screen</td>
<td>6-5, 6-6</td>
</tr>
<tr>
<td>Create</td>
<td>6-10</td>
</tr>
<tr>
<td>Special screens</td>
<td></td>
</tr>
<tr>
<td>8514/A</td>
<td>D6-3</td>
</tr>
<tr>
<td>EGA</td>
<td>D3-2</td>
</tr>
<tr>
<td>Hercules</td>
<td>D5-2</td>
</tr>
<tr>
<td>VGA</td>
<td>D4-3</td>
</tr>
<tr>
<td>specScr</td>
<td></td>
</tr>
</tbody>
</table>
xxx - Index

MultiGraphics

Overview ......................................... 23-8
String width ................................... 15-28, 21-32, 22-36, 23-20
Stroke font ................................... 15-2, 15-4
Stroke symbols .............................. 14-1, 14-3
StrWidth
  MG ............................................ 15-28
  MGIW ...................................... 21-32
  MGW ........................................ 22-36
  style16 ...................................... 17-4
Super VGA ...................................... D4-1
SuperVGA
  Detect adapter ................................ D4-1
SuperVisor
  Error handling ................................ 3-1
SVErrMsg
  Error messages .............................. 3-4, A-1
SVERRMSG.TXT
  Error messages .............................. 3-4, A-1
SVGA
  Colors ........................................ D11-8
  Line style .................................. D11-4
  Patterns .................................... D11-4
  Virtual screens ............................ D11-3
  Visible screen ............................. D11-2
Symbol
  Add ............................................ 14-24
  Delete ....................................... 14-24
  Draw ......................................... 14-32
  Drawing point .............................. 21-31, 22-35
  First ........................................ 14-30
  Height ...................................... 14-29, 21-30, 22-35
  Hot spot .................................... 14-29
  Last .......................................... 14-30
  Mouse pointer .............................. 30-25, 30-26
  Read transformation ....................... 14-32
  Width ........................................ 14-29, 21-30, 22-34
Symbol data
  GetSymbolData ................................ 14-20
  GetSymbolDataPtr ............................ 14-22
Symbol file ..................................... 14-8
Symbol formats ................................ 14-7
Symbol information ......................... 14-6
Symbol library
  Create ....................................... 14-14
Symbol mouse pointer ....................... 30-15
Symbol types .................................. 14-12, 15-10
Summary ........................................ 14-2
SymbolPointer
  MGMouse ..................................... 30-15
Screen type .................................... 6-6
SSL
  Large stroke symbols ..................... 14-8
SSS
  Small stroke symbols ..................... 14-8
Stamp
  Drawing mode ................................ 9-8
StartClockEvents
  MGClock ..................................... 28-11
StartEventHandler
  MGEv ......................................... 27-11
StartKbdEvents
  MGKbd ........................................ 29-6
StartMouseEvents
  MGMouse ..................................... 30-18
StartRecording
  MG ............................................ 24-12
StopClockEvents
  MGClock ..................................... 28-12
StopEventHandler
  MGEv ......................................... 27-12
StopEvents
  MGEv ......................................... 27-7
StopKbdEvents
  MGKbd ........................................ 29-6
StopMouseEvents
  MGMouse ..................................... 30-19
StopRecording
  MG ............................................ 24-12
StrConversionProc
  MGIO ........................................ 23-13
StrEdit
  MGIO ........................................ 23-46
StrEditor
  Data type .................................... 23-8, 23-14
Stretch
  Image ........................................ 20-19
  Image copy .................................. 20-20
String
  Draw ......................................... 15-29, 21-33, 22-37
  Read ......................................... 23-56
  Write ........................................ 23-39
String conversion
  Empty procedure ................................ 23-10
String conversion procedure
  MGIO ......................................... 23-13
String editor
  Call ........................................... 23-50
  Create ...................................... 23-49
Symbols
8514A ........................................ D6-5
Bézier type...................... 14-1, 14-4
BGI ............................................ 19-3
Bitmapped ....................... 14-1, 14-3
CGA........................................ D1-3
Contra fonts .................... 14-11
Convert ............................... 14-18
Creating .............................. 14-8
Current............... 14-2, 14-28
Data types ....................... 14-12
DeskJet ...................................... D13-5
Dispose .................................... 14-17
EGA ........................................ D3-5
Epson .......................................... D7-4
Global transformation ... 14-10, 14-19
Hercules ............................... D5-4
Load ....................................... 14-15
MCGA ........................................ D2-3
MG........................................... 14-12
Outline type .................. 14-1, 14-3
PaintJet ..................................... D12-5
Save ................................. 14-16
Set current ...................... 14-27
Set default ....................... 14-27
Standard ......................... 14-5
Stroke type .................... 14-1, 14-3
Summary ............................ 14-1
SVGA........................................ D11-5
Toshiba ................................. D10-3
Transformation ........... 14-10, 14-31
VGA........................................ D4-5
Window oriented transformation 14-10
Symbols conversion .......... 14-5
Symbols information
Get .................................. 14-17, 14-31
Symbols type
Get .......................................... 14-28
SymbolSet
MG........................................ 14-12
SymbolsType
MG........................................ 14-12, 15-10
System clock
MGClock ......................... 28-1
Time .................................. 28-14
SZL
Large Bézier symbols ........ 14-8

T
Tab

MGIO........................................ 23-19
Tabulator
Set ........................................ 23-19
Termination key
Read .................................. 23-62
Termination keys
MGIO........................................ 23-11
Read which ....................... 23-36
Set ........................................ 23-35
Standard ................................ 23-6
TermKeySet
MGIO........................................ 23-11
Text
Write ................................... 23-39
Text attributes
Position ................................ 23-4
Text column
Read .................................. 23-18
Text I/O
Capability for .................... 23-2
Summary .................................. 23-1
Text mode
MGIO........................................ 23-2
Text output
Summary .................................. 23-4
Using standard procedures .... 23-5
Text position ...................... 23-3
Set ...................................... 23-17
Text row
Read .................................. 23-18
Text string
Read .................................. 23-56
Text windows
Creating ................................ 23-15
Ticks
Definition of ......................... 28-1
From time ......................... 28-15
Per second ......................... 2-2
ticksPerSecond
MG ........................................ 2-2
Time
Converting ......................... 28-14
Current ......................... 28-15
Now ........................................ 28-3
ToggleKey
MGKbd ........................................ 29-4
ToggleKeySet
MGKbd ........................................ 29-4
Toshiba
Index

Colors ........................................ D10-4
Line style .................................. D10-2
Optimizing ................................. D10-2
Patterns .................................... D10-3
Virtual screens ........................... D10-1
Visible screen ............................ D10-1
TraceComplexOutline
MG ............................................. 10-6
TraceOutline
MG ............................................. 10-5
Tracking
Of mouse .................................... 30-3
Transformation
Font ........................................... 15-16
Fonts ......................................... 15-9
Read symbol ................................ 14-32
Symbols ................................. 14-10, 14-19, 14-31
Window oriented transformation 14-10
TransformFont ................................ 15-16
TransformPolyBezier
MG ............................................. 11-22
MGW ......................................... 22-29
TransformPolyline
MG ............................................. 11-11
MGW ......................................... 22-24
TransformShortPolyline
MG ............................................. 11-14
TransformSymbols
MG ............................................. 14-19
True color
Summary ....................................... 8-3
TURBO.EXE .................................... 1-4
TurnDir
MG ............................................. 9-14
MGW ......................................... 22-20
Typeface
Load .......................................... 15-12
Transformation ............................ 15-16
Typeface size
Epson ............................................ D7-13
Typeface weight
Epson ............................................ D7-12

U
Underlining
MGIO .......................................... 23-3
Set ............................................ 23-27
Underlining mode
Read ............................................ 23-27

V
Value
Colors ........................................ 18-1
MGCol .......................................... 18-1
vectorDevice ............................... 5-7
vert ............................................. 17-2
VESA
16 colors ..................................... D4-1
256 colors ................................... D11-1
Detect ......................................... 5-11
VESA standard ............................. H-1
VESAINFO .................................... H-1
VGA
Colors .......................................... D4-7
Line style .................................... D4-4
MGVGA ........................................ D4-1
Optimizing .................................... D4-4
Patterns ....................................... D4-4
Special screens ............................ D4-3
Virtual screens ............................. D4-2
Visible screen .............................. D4-2
Viewport
Clipping ....................................... 7-5
Copy contents .............................. 13-6
Get current .................................... 9-16
Pattern fill .................................... 12-4
Read .......................................... 21-18, 22-21
Set ............................................. 21-18, 22-20
Set current .................................... 9-15
Summary ....................................... 9-3
virtScr
Screen type .................................... 6-6
Virtual Memory ............................. 1-5
Automatic selection ....................... B-3
Images ......................................... 20-4
Summary ....................................... B-1
Virtual screen ................................ 6-4, 6-6
Create ......................................... 6-7
Image ......................................... 20-26
Virtual screens
8514/A ......................................... D6-2
Canon Laser .................................. D14-3
CGA ............................................. D1-2
DeskJet ......................................... D13-3
EGA ............................................. D3-2
Epson .......................................... D7-2

MultiGraphics & Borland Pascal/Turbo Pascal  DATABITEN AB
MultiGraphics

Hercules.............................. D5-2
HP LaserJet.......................... D8-2
MCGA.................................. D2-1
PaintJet............................... D12-2
SVGA.................................. D11-2
Toshiba............................... D10-1
VGA.................................... D4-2
Visible screen.......................... 6-3, 6-6
8514/A................................. D6-2
Canon Laser.......................... D14-2
CGA..................................... D1-1
DeskJet................................. D13-2
EGA...................................... D3-1
Epson.................................. D7-2
Hercules............................... D5-1
HP LaserJet.......................... D8-2
HPGL.................................... D9-2
MCGA.................................. D2-1
PaintJet............................... D12-2
SVGA.................................. D11-2
Toshiba............................... D10-1
VGA.................................... D4-2
visiScr
Screen type............................ 6-6
VM
Expanded memory...................... B-2
Extended memory...................... B-2
Large images.......................... 20-4
Virtual memory management.......... B-2
Virtual Memory Manager.............. 1-5
W
Wait
MGIO.................................... 23-45
On keystroke.......................... 23-45
WaitNoFlicker
MG........................................ 5-24
Warning sounds
Make.................................... 23-33
Set........................................ 23-32
Used?................................... 23-32
White-on-black
Color selection........................ 8-7
Current.................................. 8-19
Set........................................ 8-19
Width
Attribute characters.................. 23-19
Attribute string....................... 23-20
Characters............................. 15-26, 21-31, 22-35
Convert from world coord21-12, 22-14
Current screen.......................... 6-16
Image.................................... 20-27
Line....................................... 20-27
Line read.................................. 11-8
String................................. 15-28, 21-32, 22-36
Symbol................................. 14-29, 21-30, 22-34
Window................................ 7-15, 7-20
Window
Adjust.................................. 7-10
Capability............................... 7-4
Clear..................................... 7-22, 23-21
Create................................... 7-8
Creating text........................... 23-15
Current................................. 7-4, 7-17
Dispose................................. 7-11
Exists.................................... 7-18
For mouse............................... 30-22
Height................................. 7-16, 7-20
Increase capability.................... 7-14
Is a screen window?.................... 7-18
Mode..................................... 7-21
Overlap................................. 7-3
Overview............................... 7-1
Scroll................................... 23-23
Set current............................. 7-16
Set screen window..................... 7-17
Type..................................... 7-6
Width.................................... 7-15, 7-20
With handle............................. 7-12
With world capability.............. 21-7, 22-9
X position............................. 7-15, 7-19
Y position............................. 7-15, 7-20
Window columns....................... 23-16
Window coordinate.................. 21-5, 22-5
Window coordinates............... 7-3
x from world coord21-11, 22-13
y from world coord21-12, 22-14
Window handle........................ 7-2
Get........................................ 7-12
Return................................... 7-19
Window handles
At position......................... 7-13, 21-8, 22-10
Window mode.......................... 7-2, 9-1
Contra world mode.................. 21-4, 22-4
Get........................................ 7-22
Set........................................ 7-21
Window position...................... 21-5, 22-5
Window rows.......................... 23-23
Window units.......................... 21-5, 22-5
<table>
<thead>
<tr>
<th>WindowMode</th>
<th>Data type</th>
<th>7-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>Palette</td>
<td>18-6</td>
</tr>
<tr>
<td></td>
<td>Set palette</td>
<td>18-6</td>
</tr>
<tr>
<td>WinExist</td>
<td>MG</td>
<td>7-18</td>
</tr>
<tr>
<td>WinHandle</td>
<td>MG</td>
<td>7-12</td>
</tr>
<tr>
<td>WinHeight</td>
<td>MG</td>
<td>7-15</td>
</tr>
<tr>
<td>WinOK</td>
<td>MG</td>
<td>7-16</td>
</tr>
<tr>
<td>WinWidth</td>
<td>MG</td>
<td>7-12</td>
</tr>
<tr>
<td>WinX</td>
<td>MG</td>
<td>7-15</td>
</tr>
<tr>
<td>WinY</td>
<td>MG</td>
<td>7-15</td>
</tr>
<tr>
<td>Word</td>
<td>Read</td>
<td>23-57</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>23-40</td>
</tr>
<tr>
<td>WordCheck</td>
<td>MGIO</td>
<td>23-54</td>
</tr>
<tr>
<td>World coordinate</td>
<td></td>
<td>22-5</td>
</tr>
<tr>
<td>Integer based</td>
<td></td>
<td>21-5</td>
</tr>
<tr>
<td>World coordinate mode</td>
<td></td>
<td>21-8, 22-10</td>
</tr>
<tr>
<td>World coordinates</td>
<td></td>
<td>21-5, 22-5</td>
</tr>
<tr>
<td>Absolute limits</td>
<td></td>
<td>21-1</td>
</tr>
<tr>
<td>MGIW</td>
<td></td>
<td>22-1</td>
</tr>
<tr>
<td>MGW</td>
<td></td>
<td>22-1</td>
</tr>
<tr>
<td>Read</td>
<td></td>
<td>21-10, 22-12</td>
</tr>
<tr>
<td>Read limits</td>
<td></td>
<td>21-11, 22-12</td>
</tr>
<tr>
<td>Set</td>
<td></td>
<td>21-10, 22-11</td>
</tr>
<tr>
<td>Uniform</td>
<td></td>
<td>22-4</td>
</tr>
<tr>
<td>x from device coordinates</td>
<td></td>
<td>21-13, 22-15</td>
</tr>
<tr>
<td>y from device coordinates</td>
<td></td>
<td>21-14, 22-15</td>
</tr>
<tr>
<td>world height</td>
<td></td>
<td>21-14, 22-16</td>
</tr>
<tr>
<td>World mode</td>
<td>Data type</td>
<td>21-6</td>
</tr>
<tr>
<td></td>
<td>Data types</td>
<td>22-8</td>
</tr>
<tr>
<td>MGIW</td>
<td></td>
<td>21-4, 21-6</td>
</tr>
<tr>
<td>MGW</td>
<td></td>
<td>22-4, 22-8</td>
</tr>
<tr>
<td>Set</td>
<td></td>
<td>21-9, 22-11</td>
</tr>
<tr>
<td>World position</td>
<td></td>
<td>22-5</td>
</tr>
<tr>
<td>Integer based</td>
<td></td>
<td>21-5</td>
</tr>
<tr>
<td>World units</td>
<td></td>
<td>22-6</td>
</tr>
<tr>
<td>Integer based</td>
<td></td>
<td>21-5</td>
</tr>
<tr>
<td>World width</td>
<td></td>
<td>21-14, 22-16</td>
</tr>
<tr>
<td>From device width</td>
<td></td>
<td>21-14, 22-16</td>
</tr>
<tr>
<td>WorldHeight</td>
<td>MGIW</td>
<td>21-14</td>
</tr>
<tr>
<td></td>
<td>MGW</td>
<td>22-16</td>
</tr>
<tr>
<td>WorldMode</td>
<td>MGW</td>
<td>22-8</td>
</tr>
<tr>
<td>WorldWidth</td>
<td>MGIW</td>
<td>21-14</td>
</tr>
<tr>
<td></td>
<td>MGW</td>
<td>22-16</td>
</tr>
<tr>
<td>WorldY</td>
<td>MGIW</td>
<td>21-14</td>
</tr>
<tr>
<td></td>
<td>MGW</td>
<td>22-15</td>
</tr>
<tr>
<td>WrChar</td>
<td>MGIO</td>
<td>23-39</td>
</tr>
<tr>
<td>WrFixReal</td>
<td>MGIO</td>
<td>23-43</td>
</tr>
<tr>
<td>WrInt</td>
<td>MGIO</td>
<td>23-41</td>
</tr>
<tr>
<td>Write</td>
<td></td>
<td>23-40</td>
</tr>
<tr>
<td>Characters</td>
<td></td>
<td>23-39</td>
</tr>
<tr>
<td>Fix Real</td>
<td></td>
<td>23-43</td>
</tr>
<tr>
<td>In graphics mode</td>
<td></td>
<td>23-5</td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td>23-41</td>
</tr>
<tr>
<td>LongInt</td>
<td></td>
<td>23-42</td>
</tr>
<tr>
<td>New line</td>
<td></td>
<td>23-38</td>
</tr>
<tr>
<td>Real</td>
<td></td>
<td>23-42</td>
</tr>
<tr>
<td>String</td>
<td></td>
<td>23-39</td>
</tr>
<tr>
<td>Word</td>
<td></td>
<td>23-40</td>
</tr>
<tr>
<td>WriteLn</td>
<td></td>
<td>23-5</td>
</tr>
<tr>
<td>In graphics mode</td>
<td></td>
<td>23-5</td>
</tr>
<tr>
<td>WrLn</td>
<td>MGIO</td>
<td>23-38</td>
</tr>
<tr>
<td>WrLong</td>
<td>MGIO</td>
<td>23-42</td>
</tr>
<tr>
<td>WrReal</td>
<td>MGIO</td>
<td>23-42</td>
</tr>
<tr>
<td>WrStr</td>
<td>MGIO</td>
<td>23-39</td>
</tr>
<tr>
<td>WrWord</td>
<td>MGIO</td>
<td>23-40</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>21-11, 22-13</td>
</tr>
<tr>
<td>x coordinates</td>
<td></td>
<td>21-11, 22-13</td>
</tr>
</tbody>
</table>
X position
  Get current ........................................... 9-12
  Read .................................................... 21-16, 22-18

XMS
  XMSVM ................................................. B-2

XMSVM
  Extended memory ........................................ B-2
  XMS memory ............................................. B-2

xonxoff
  Flow control ........................................... 26-4

XOR
  Drawing mode ........................................... 9-9

Y
  y coordinates
    From world ........................................ 21-12, 22-14

Y position
  Get current ........................................... 9-13
  Read .................................................... 21-17, 22-18

Z
  Zoom
    Copy of image ..................................... 20-20
    Image ................................................. 20-19
  Zoom screen ........................................... 6-11, 13-7
  ZoomCopyImage
    MGImg ............................................... 20-20
  ZoomImage
    MGImg ............................................... 20-19
  ZoomVirtScr
    MG ..................................................... 6-11, 13-7