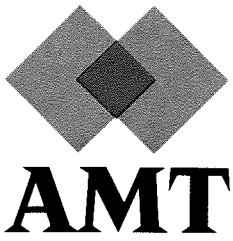


DAP Series

Low-level Graphics Library

(enhanced)

GRALIB

(man117.01)

we consider you would not be following good (or at least normal) coding practice. The words 'you should (not) ... ' elsewhere in the body of the text are meant to give you a similar caution.

typographical conventions

The following typographical conventions are used in this manual:

- Names of variable, commands, functions, subroutines and files mentioned in the text are shown in **bold type face**
- Computer screen or hard copy output is shown in a box:

```
This is an example of screen output
```

- Any input that you would type is shown in **bold type face**.

  Occasionally, what you have to type in is shown boxed, as well as being shown in **bold typeface**
- Text that would be replaced by other text in what you type in or what the computer outputs is shown in *italics*.

  For example, you might be asked to type the command:

  save *name*

  When you come to type the command you would replace *name* with the name of the file into which you wanted to save whatever was involved.

  Similarly, a host screen display might be shown as:

```
Version n.m with SCSI HCU link
MCU code size 512 Kbytes, array size 4 Mbytes
TWON>
```

  whereas, in what you would actually see on your screen, *n.m* would replaced by a number combination, such as 3.1
- If you are asked to press a particular key on the keyboard, that key will printed in capital letters and will be enclosed in angled brackets. For example:

  <RETURN>

  is asking you to press the Return key
- If you are asked to press one key whilst holding down another key, both keys will be enclosed in angled brackets, with the to-be-held-down key given first and the keys joined by a '-'. For example:

  <CONTROL-Z>

  is asking you to hold down the Control key and press the 'Z' key.

Similarly:

<CONTROL-SHIFT-Q>

is asking you to press and hold down the Control key, then press and hold down either Shift key, and then press the 'Q' key

command syntax

The syntax for a command specifies optional and alternative sub-items in the command as:

[ ] You don't need to include any of the item(s) enclosed in square brackets, but if you do, then you can only include one

{ } You must have one – and only one – of the items enclosed in braces

. . . You can repeat the item (and its delimiter, if appropriate) preceding an ellipsis zero or more times; that is, the item can occur one or more times

For example, a hypothetical command might be specified as:

$$\left\{\begin{matrix}\mathbf{ab}\\ \mathbf{ac}\\ \mathbf{ad}\end{matrix}\right\}\left[\begin{matrix}\textit{option}\left[,\ \textit{option}\ \ldots\right]\\ \textit{filename}\end{matrix}\right]$$

Possible variations of the command include:

ab

ab *option*

ab *option1, option2, option3*

ab *filename*

ac *option1, option2*

and so on, where *option*, *option1*, *option 2*, *option3* and *filename* would be defined as appropriate to the command.

Table of Contents

Chapter 1

Introduction and usage

Your DAP system may include hardware for driving a suitable high resolution colour monitor. If such hardware is installed, you can use the AMT graphics library **gralib**, supplied with the basic DAP software, to construct images and display them on your monitor.

The image displayed on the monitor is made up of 1024^2 *pixels* (or picture elements). If your DAP system is fitted with a VO-24 video output coupler, you can define pixels in *direct colour mode* as three-channel (Red, Green Blue) composite values, using 1, 2, 4 or 8 bits per colour channel which gives 3, 6, 12 or 24 bits per pixel. Direct colour is also referred to as *true colour*.

If you have a VO-8 or an earlier DPIO video output system, pixel values defined with 1, 2, 4 or 8 bit precision are used to select colours out of the current *palette* or colour *look-up table*. This is known as *mapped colour mode* (sometimes referred to as *false colour mode*), since the look-up table provides a mapping between the 16 million possible colours and the limited number of colours available on the screen at the same time with the VO-8 or DPIO hardware.

You can change the look-up table at any time, to one of the AMT-supplied tables, or to one of your own making.

the display buffer

You construct the image you want to display in a data area within your program block. This part of array memory allocated to your program is referred to in this manual as the *display buffer* (*screen buffer* in previous AMT documentation). A typical declaration in a FORTRAN-PLUS enhanced program for a mapped-colour display buffer might be:

```
INTEGER*1 IMAGE(*1024,*1024)
```

You can construct an image in your display buffer either by calling one or more of the AMT-supplied routines in **gralib** (such as **amt_gra_put_lines**), or by explicitly updating the pixel values in the display buffer using your own code, or by some combination of the two techniques. You only need to transfer the image from the display buffer to the hardware framestore when you want to change the picture displayed on the monitor. The **gralib** routines let you specify such an update either on a one-off basis (by using routine

amt_gra_put_frame), or repeatedly (using **amt_gra_start_sequence**). You are strongly advised to define your display buffer in a *common block*, to make sure that the buffer contents are preserved between calls to the routines.

using co-ordinates

With some **gralib** routines you will need to specify the co-ordinates of individual pixels. The x co-ordinates (left to right on the screen) and the y co-ordinates (top to bottom on the screen) are integers in the range 0 to 1023. The top left corner of the screen has co-ordinates x=0, y=0; the bottom right corner has x=1023, y=1023; and so on.

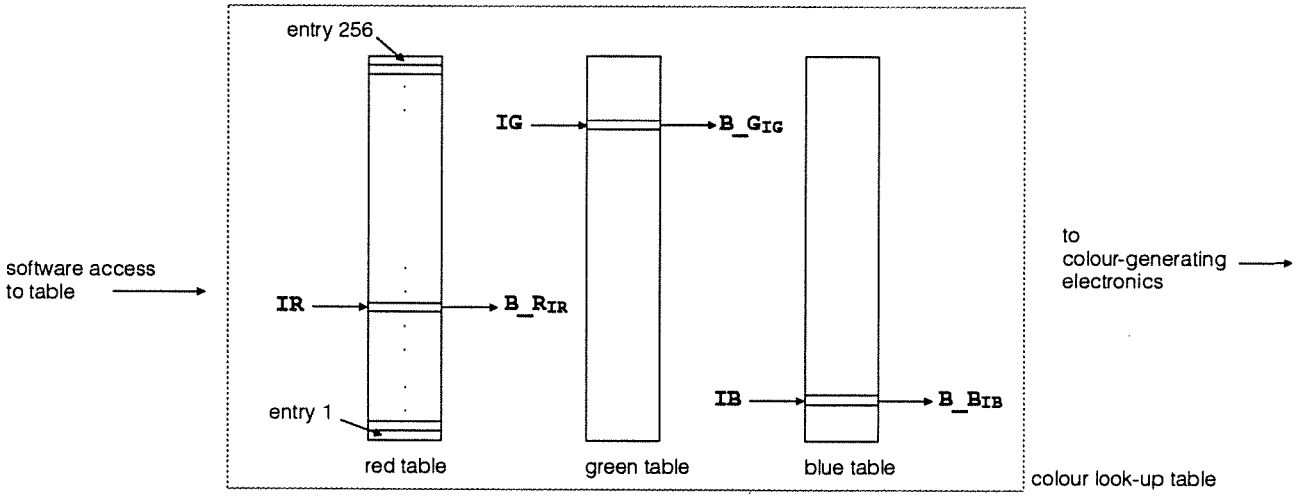
The first row of pixels on the screen corresponds to *display-buffer*(1,), where *display-buffer* is the name you have declared for the variable holding your display buffer. Similary, the n^{th} row of pixels on the screen corresponds to *display-buffer*(*n*,), and the n^{th} column of pixels on the screen corresponds to *display-buffer*(,*n*).

1.1 Colour modes and generation

how colour is generated

The DAP display hardware uses three 256-entry tables as a colour look-up table. There is one table for each of the 3 primary colours, red, green, and blue. The tables provide the link between the pixel values the software handles and the colours displayed on the DAP monitor screen.

The diagram below illustrated the operation of the look-up table. The entries in the red, green and blue sub-tables are accessed by separate pixel-controlled pointers or indices I_R, I_G and I_B. The values held in the 3 tables at the three access points, B_R_{IR}, B_G_{IG} and B_B_{IB}, specify the brightnesses of the primary colours to be displayed on the DAP monitor screen.



Each entry in each table can hold a value in the range 0 to 255; 255 specifies maximum brighness in the relevant primary colour, 0 specifies zero brightness.

The values held in the 3 tables specify what colour will be displayed for a given index into each table, and you can load different sets of values into the 3 tables. These sets of values – *software* look-up tables – give you great control over the colours displayed. You can use one of the AMT-supplied standard software look-up tables, but you can also generate your own tables, or modify one of the AMT tables, and load that.

How you specify colours in your software depends on whether you are working with three channel *direct* colour using the VO-24, or single channel *mapped* colour using the VO-8 coupler or the earlier DPIO coupler. You will have to supply most **gralib** routines with a colour value (or in some cases a set of colour values), as a parameter. These colour values are 32 bit integers, only the least significant *pxl* bits being used, where *pxl* is the number of bits per pixel.

direct colour mode

Using direct colour mode, you access each sub-table in the look-up table separately. The indices I_R, I_G and I_B of the diagram opposite are coded into one pixel value, using a pixel length *pxl* of 3, 6, 12 or 24 bit precision (1, 2, 4 or 8 bits for each of I_R, I_G and I_B).

mapped colour mode

In mapped colour mode, you use the same pointer to access all three tables. In terms of the diagram on the opposite page, you only supply one index, **I** say, and $I=I_R=I_G=I_B$. Each pixel value in mapped colour mode is defined with a *pxl* of 1, 2, 4 or 8 bit precision.

Note that although the three channel mode is described as direct mode, it too makes use of the mappings built into the look-up table. Whereas in mapped mode, the single index into the look-up table controls all three primary displayed colours, in direct mode any one of the 3 indices into the table can only control one primary.

either colour mode

When you initialise the **gralib** routines, a default colour look-up table, giving 256 greyscale values, is automatically loaded.

setting pixel values directly

To set a value into a pixel directly, the value you supply has to match the declared type of your display buffer. **INTEGER*1** variables (for mapped mode) are *signed* quantities and can take values in the range -128 to 127, while **INTEGER*3** variables (for direct mode) take values in the range -8388608 to 8388607. Hence, for mapped mode, to specify a brightness in the range 128 to 255, you need to specify a value in the range -128 to -1.

For other precisions, where the display buffer has been declared as if it were an array of **LOGICAL** bit planes, you have to set individually each bit of each pixel you want to change.

When you are using **gralib** routines, you are insulated from the problem of specifying colours in signed integer format. For the routines, all colours are held in the lower *pxl* bits of a 32-bit integer variable; conversion to signed integer format is carried out for you by **gralib**.

1.2 Display buffer

In earlier versions of **gralib**, the display buffer (or screen buffer as it was then called) was fixed at the full-screen size of 1024 by 1024 pixels.

The version of **gralib** available from DAP basic software release 3.3 lets you define a display buffer smaller than full-screen. A new **gralib** command, **amt_gra_define_image**, lets your specify the image to be displayed as different from full-screen. The defined image can be the whole of a sub-screen size display buffer, or just part of a normal full-screen display buffer

A major advantage of a smaller-than-screen-size buffer is that you no longer have to reserve at least 1 Mbytes (1024 x 1024 8-bit pixels) for your display buffer.

1.3 Summary

In summary, the AMT graphics library is a set of routines that you can call from FORTRAN-PLUS or APAL code, which let you:

- Nominate a data area within your DAP program block as a display buffer
- Specify a colour look-up table
- Draw characters, dots, lines and so on into the display buffer
- Magnify an existing image
- Output the display buffer to the screen on a one-off or regular basis
- Define an image for display smaller than your display buffer

As an aid to debugging a graphics program you are provided with two routines:

- **amt_gra_turn_on_error_messages**
- **amt_gra_turn_off_error_messages**

which control messages that can be sent to the host screen as a **Trace** report when the library routines detect possible error conditions. Error messages are output by default and any unexpected results from your graphics routines should prompt you to check these messages.

This version of the library accompanies version 3.3 or higher of the DAP basic software, and is designed for use with any DAP-series machine; that machine can be connected to either a Sun or a Digital VAX/VMS system host. There are differences between this version of **gralib**, and the one available with release 3.2 or earlier of the DAP software: see section 1.4 for more details of the differences.

The DAP basic software supports multi-programming, and several programs can run simultaneously on a time-slice basis. Provided no other program has issued an **amt_gra_init_graphics** command without a following **amt_gra_stop_graphics** command, if your program issues an **amt_gra_init_graphics** command, you will get access to the graphics display system. If another program is already using graphics when your program issues **amt_gra_init_graphics**, the system returns a non-zero error code, which your program should trap and deal with appropriately. See pages 18, 35 and 49 later in the manual for more details of **amt_gra_init_graphics**, **amt_gra_stop_graphics** and error messages from **amt_gra_init_graphics** respectively.

A DAP simulator is also supplied with your basic software; DAP programs that include calls to **gralib** will work on the simulator but produce no screen output. Graphics routines running on a simulator will return the normal error messages and codes (except that errors associated with the unavailability of a monitor are not sent).

1.4 Differences from earlier version of **gralib**

The current version of **gralib** has been designed to use with release 3.3 or higher of the DAP basic software, which includes FORTRAN-PLUS enhanced, the unconstrained version of FORTRAN-PLUS.

With this latest version of **gralib** you do not, in general, have to concern yourself as to how your display buffer and other images are mapped onto the array store of the target DAP. In earlier **gralib** versions, you had to partition your buffer and images into *tiles*, each of size *ES* x *ES* pixels, where *ES* is the edge size of the target DAP. Graphics source code written for release 3.2 or earlier will still work under release 3.3, but you have to recompile the source if you want to use it with code written in FORTRAN-PLUS enhanced.

extra routines in the new gralib

Two new commands have been added to the new version of **gralib**: **amt_gra_define_image** and **amt_gra_magnify** – the second of which has been added as routine **magnify** to the old version of **gralib** included in release 3.3. **magnify** is documented in appendix B on page 53 of this manual.

Names of routines in the new **gralib** all start with **amt_gra_**, names in the old **gralib** had no such prefix. The old routines are supported in the new **gralib**, and you can mix the two. No details are given in this manual of the old routines (except as mentioned above); they are documented in [5], the AMT manual *DAP Series: Low Level Graphics Library* (man017).

1.5 Using the library on different DAP machines

Because you can declare your images and display buffer independently of the edge size of the target DAP, you can write graphics (and other) source code without regard to DAP edge-size. The only time when edge-size is important is when you compile and link your code.

1.6 Compilation and linking procedure

1.6.1 In a Sun UNIX environment

You can link the graphics library into a program by using the **-l** option to either **dapa** or **dapf** (see [2], *DAP Series: Program Development under UNIX* (man003), for more details).

For example, a FORTRAN-PLUS source program in a file **picture.df** can be compiled and linked with the AMT-supplied **gralib** graphics routines, and the object code put into a DOF file **picture** by executing the command:

```
host# dapf -o picture picture.df -l gralib
```

If you want to port FORTRAN-PLUS or APAL code containing calls to graphics subroutines to a DAP of different edge size, then you have to recompile and relink the code. Provided you have not declared any of your images or display buffer in terms of DAP edge-size, you do not need to change your source code. If you do use DAP edge-size in your image of buffer declarations, then you will need to use the **#if** facility. See section 2.3.3 in [2] for more details.

1.6.2 In a VAX/VMS environment

You can link the graphics library into a program by including it in the list of input files to the **DLINK** command (see [3], *DAP Series: Program Development Under VAX/VMS* (man004) for further details). In release 3.3V of the basic software, two versions of the graphics library are supplied, **GRALIB5** for DAP 500, and **GRALIB6** for DAP 600; when you link the graphics library into your program you need to specify the appropriate version of **GRALIB**.

To compile and link the DAP program in the file **PICTURE.DFP** to run on a DAP 600, you can use the following commands:

```
$ DFORTRAN/DAPSIZE=64 PICTURE
$ DLINK/DAPSIZE=64 PICTURE,SYS$LIBRARY:GRALIB6/LIBRARY
```

To compile and link code in file **IMAGES.DFP** to run on a DAP 500 the above commands would be:

```
$ DFORTRAN/DAPSIZE=32 IMAGES
$ DLINK/DAPSIZE=32 IMAGES,SYS$LIBRARY:GRALIB5/LIBRARY
```

As an alternative to specifying that the graphics routines which program **PICTURE** references are to be found in library **SYS$LIBRARY:GRALIB6.DLB**, or the routines for **IMAGES** are in library **SYS$LIBRARY:GRALIB5.DLB** you can define the logical name **DAP*n*_LIBRARY** by using the command:

```
$ DEFINE DAPn_LIBRARY SY$LIBRARY:GRALIBn
```

where *n* is 5 (for DAP 500) or 6 (for DAP 600). This will cause **DLINK** to search **GRALIB*n*** automatically for unsatisfied external references. If you are going to use **GRALIB*n*** frequently, you can insert the above **DEFINE** into your **LOGIN.COM** file. If there are several DAP users on the system, linked to a DAP 600 say, the system manager could include the command:

```
$ DEFINE/SYSTEM DAP6_LIBRARY SY$LIBRARY:GRALIB6.DLB
```

into the site system start-up command file which would give all users automatic access to the library.

Similarly, the command:

```
$ DEFINE/SYSTEM DAP5_LIBRARY SYS$LIBRARY:GRALIB5.DLB
```

would achieve the same thing for a DAP 500 system.

linking several DAP libraries

If you want **DLINK** to search more than one library automatically, then in addition to defining **DAP*n*_LIBRARY** you can define one or more of the following logical names:

```
DAPn_LIBRARY_1
DAPn_LIBRARY_2
...
```

DLINK will scan each library specified by these names in turn, stopping at the first value of *m* for which **DAP*n*_LIBRARY_*m*** is not defined. Note that each logical name should specify only one DAP library.

For example, if a program to run on a DAP 600 uses routines in both **GRALIB** and **DSPLIB**, then if you have already

defined **DAP6_LIBRARY** to be **GRALIB6**, and you use the command:

```
$ DEFINE DAP6_LIBRARY_1 SYS$LIBRARY:DSPLIB6
```

when **DLINK** is called it will scan both **GRALIB6** and **DSPLIB6** libraries.

On a system that has available both DAP 500 and DAP 600, then both **DAP5_LIBRARY** and **DAP6_LIBRARY** can be defined, and users will pick up the appropriate version of **GRALIB** when they specify **DAPSIZE** in their **DFORTRAN** and **DLINK** commands.

If you want to port FORTRAN-PLUS or APAL code containing calls to graphics subroutines to a DAP of different edge size, then you have to recompile and relink the code. Provided you have not declared any of your images or display buffer in terms of DAP edge-size, you do not need to change your source code. If you do use DAP edge-size in your image or buffer declarations, then you will need to use the **#if** facility. See section 2.2.5.3 in [3] for more details.

Chapter 2

Details of routines

2.1 Introduction

This chapter gives a short description of each routine, and includes an example of a typical calling sequence. Any DAP program that calls any of the routine should have declarations similar to those given in these examples. Declarations have to precede the call, but can appear anywhere before the call itself, either together or separately.

The colour construction functions **amt_gra_RGB_val** and **amt_gra_RGB_vals** are also included in the following descriptions.

The routines should return an error code of zero in the **IERR** parameter. Any other value indicates an error – a list of possible error codes is given in appendix A starting on page 49.

The routines are listed in alphabetical order in the main part of this chapter; the list that follows lists the routines in groups of related function.

Routine	*Action*
amt_gra_init_graphics	Obtains graphic resources
amt_gra_stop_graphics	Releases the graphics resources
amt_gra_set_colour_regime	Specifies how pixels from different graphic objects are to be combined in the display buffer
amt_gra_set_lut	Loads an AMT-supplied look-up table
amt_gra_put_lut	Loads a look-up table from array store
amt_gra_get_lut	Copies the current look-up table into array store
amt_gra_change_screen	Changes the area of array memory to be used as display buffer
amt_gra_put_frame	Outputs a single image to DAP monitor screen
amt_gra_start_sequence	Starts the repeated output of images to DAP monitor screen
amt_gra_stop_sequence	Stops the repeated output of images to DAP monitor screen
amt_gra_clear_screen	Clears the image in the display buffer and replaces it with a background colour
amt_gra_put_dots	Plots dots

Routine	*Action*
amt_gra_put_rectangles	Draws rectangles
amt_gra_put_lines	Draws lines
amt_gra_put_wide_lines	Draws wider lines and parallelograms
amt_gra_init_font_*fontname*	Loads a font
amt_gra_put_characters	Draws a character string
amt_gra_rasterop	Copies to part of the display buffer an image from another part of the buffer or from another part of array memory
amt_gra_copy_image	Copies to part of the display buffer a tile-aligned image from another part of the buffer or from another part of array memory
amt_gra_magnify	Magnifies an image
amt_gra_define_image	Specifies image attributes that are used when the framestore hardware outputs to the DAP monitor screen
amt_gra_turn_on_error_messages	Turns on error messages
amt_gra_turn_off_error_messages	Turns off error messages

Colour construction function	*Action*
amt_gra_RGB_val	Generates a 24 bit colour value from its red, green and blue constituents
amt_gra_RGB_vals	Generates a set of 24 bit colour values from their red, green and blue constituents

The full-screen-size image is 1024 by 1024 pixels; you normally don't have to concern yourself with processing individual *ES* x *ES tiles* (where *ES* is the edge-size of the target DAP). However, a few **gralib** routines only handle images made up of complete tiles.

declaration of display buffer

To avoid repetition in this manual, in the various fragments of code the display buffer is specified as **INTEGER*** *n*. For 8 bit mapped colour this should read **INTEGER*1**, whereas for 24 bit direct colour it should read **INTEGER*3**. For other pixel lengths the display buffer has to be declared as if each tile were an array of logical bit-planes (see section 1.1 on page 3).

routines handle co-ordinates in vector or matrix arguments

Many of the **gralib** routines take as some of their arguments, parameters specifying the co-ordinates and colours of the objects to be drawn on the screen – and specifying the accompanying filtering-out masks. In all such cases you can specify such co-ordinates and so on as components in either vectors or matrices, although you can't use both in the one routine.

You cannot use scalars for the required parameters; if you only want one graphic object, say a rectangle, then you would have to declare a set of one element vectors – or matrices!

2.2 amt_gra_change_screen

amt_gra_change_screen nominates a new display buffer, perhaps to let you switch between ready-prepared images.

Typical calling sequence:

```
INTEGER*n IMAGE2(*1024,*1024)
:
INTEGER IERR
:
CALL AMT_GRA_CHANGE_SCREEN(IMAGE2,IERR)
```

The routine takes the arguments:

Argument	*Description*
IMAGE2	The new display buffer
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See appendix A starting on page 49 for details of the meaning of a non-zero error code

If you have called **amt_gra_start_sequence** you should call **amt_gra_stop_sequence** before changing the location of the display buffer.

As with **amt_gra_init_graphics**, the new display buffer can be smaller than a full screen, although you should also call **amt_gra_define_image** (see page 13) if you want to declare a smaller-than-full-screen display buffer.

2.3 amt_gra_clear_screen

amt_gra_clear_screen sets the whole of the nominated display buffer to a uniform background colour.

Typical calling sequence:

```
INTEGER ICOL, IERR
:
CALL AMT_GRA_CLEAR_SCREEN(ICOL,IERR)
```

The routine takes the arguments:

Argument	*Description*
ICOL	The pixel value of the cleared screen
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

amt_gra_clear_screen sets the whole of your nominated display buffer to a colour specified by the pixel value, that is, the least significant *pxl* bits of **ICOL** (*pxl* being 1, 2, 4 or 8 for mapped colour; or 3, 6, 12 or 24 bits for direct colour).

2.4 amt_gra_copy_image

amt_gra_copy_image copies some or all of the bit-planes of its input image into selected bits in the nominated display buffer.

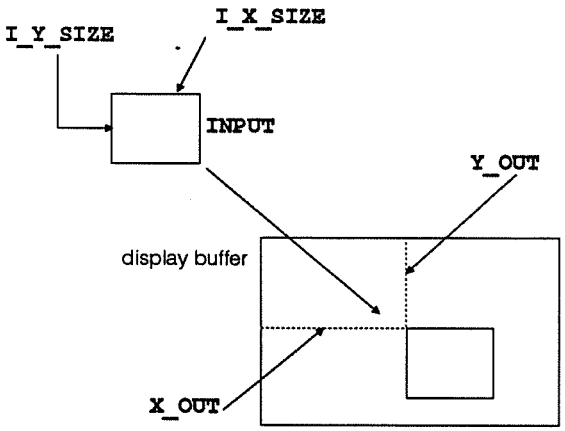
Typical calling sequence:

```
LOGICAL INPUT(*I-X-SIZE,*I-Y-SIZE,W)
:
INTEGER W,B_IN,BN,X_OUT,Y_OUT,B_OUT
:
CALL AMT_GRA_COPY_IMAGE(INPUT,B_IN,BN,X_OUT,Y_OUT,B_OUT,IERR)
```

The routine takes the following arguments:

Argument	*Description*
INPUT	The input image. **I_X_SIZE** is the horizontal size of the image to be copied, is measured in pixels, and is an integer multiple of *ES*, the edge-size of the target DAP. **I_Y_SIZE** is the vertical size of the image to be copied, is measured in pixels, and is an integer multiple of *ES*, the edge-size of the target DAP. **W** is the wordlength, in bits, of the pixels in the input image
B_IN	The bit number of the pixels in the input image from where the copying is to start
BN	The number of bits per pixel to be copied
X_OUT	The column address, in pixels, of the left edge of the receiving region in the display buffer (an integer multiple of *ES*)
Y_OUT	The row address, in pixels, of the top edge of the receiving region in the display buffer (an integer multiple of *ES*)
B_OUT	The bit number of the pixels in the display buffer at which the copying is to start
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 51 in appendix A for details of the meaning of a non-zero error code

The routine copies an input image (held in a data area of your program block) into a region of the same size in the display buffer, over-writing the existing pixels in that part of the buffer. Unlike **amt_gra_rasterop** (described in section 2.17 on page 29) the edges of both the input and output regions are constrained to lie on tile boundaries.



Also unlike **amt_gra_rasterop**, the number of bits in the data being copied across is not constrained to *pxl* bits, and you can specify a range of consecutive bit planes within the input image to be copied to a range of consecutive bit planes within the corresponding part of the display buffer.

The sketch in the margin illustrates some aspects of what the routine does.

The location of the input image is in general different from that of the buffer.

Bits within a pixel are numbered 0 to *pxl*–1, bit 0 being the most significant, where *pxl* is the number of bits in the pixel (in the typical calling sequence above, **W** for input image pixels; whatever was declared in the relevant call to **amt_gra_init_graphics** or **amt_gra_change_screen** for the display buffer pixels).

So, for example, if *pxl*=8 for **INPUT** and display buffer, **B_IN**=2, **B_OUT**=5 and **BN**=3, the effect is to copy bits 2 to 4 in each pixel in the input image to bits 5 to 7 in the corresponding pixels in the display buffer. The result is that the least significant 3 bits of the selected 8 bit pixels in the display buffer are over-written, but their most significant 5 bits are unaffected.

There are no restrictions on the value of **W**, that is, non-standard wordlengths may be used – perhaps 32 bits, 10 bits, or 1 bit.

caution

You can use the display buffer itself as the input image, but if you do, you should make sure that the input and output regions do not overlap – if they do, the effect is undefined.

2.5 amt_gra_define_image

amt_gra_define_image records image attributes for subsequent use by the framestore hardware, when a picture is output to screen, as directed by the commands **amt_gra_put_frame** and **amt_gra_start_sequence**.

A typical calling sequence:

```
 INTEGER WE_SIZE,NS_SIZE,PXL,OFFSET_B_OUT,IMG_X,IMG_Y,FRM_X,FRM_Y,
&        DISPLAY_OPT,IERR
 :
 LOGICAL COL_MAJOR_FLG,PACKED_FLG
 :
 CALL AMT_GRA_DEFINE_IMAGE(WE_SIZE,NS_SIZE,PXL,OFFSET_B_OUT,
&             COL_MAJOR_FLG,PACKED_FLG,IMG_X,IMG_Y,FRM_X,FRM_Y,DISPLAY_OPT,IERR)
```

All the routine's arguments are scalars. They are listed on the next page.

Argument	*Description*
WE_SIZE	The width of the image area you want to define in the nominated display buffer, in units of tiles
NS_SIZE	The height of the image area you want to define in the nominated display buffer, in units of tiles
PXL	The number of bits per pixel in the image area to be defined. Note: 24-bit working is only available on DAPs fitted with a V0–24 24-bit graphics board
OFFSET_B_OUT	The bit number of the pixels in framestore to which transfer from the display buffer is to start (for more detail, see the paragraphs below this table). You will normally specify an **OFFSET_B_OUT** of 0
COL_MAJOR_FLG	If this logical flag is set .TRUE. then **gralib** assumes that tiles of the image area to be defined are in column-major order, otherwise it assumes row-major ordering
PACKED_FLG	If this logical flag is set .TRUE. then **gralib** assumes that the image area to be defined is the whole of the display buffer, otherwise it assumes that the display buffer is screen-sized – that is, 1024 by 1024 pixels
IMG_X	The x co-ordinate of the first tile in the image area to be defined, measured in units of tiles and measured with respect to the top-left-hand corner of the display buffer. If **PACKED_FLG** is .TRUE., then **IMG_X** has to be 0
IMG_Y	The y co-ordinate of the first tile in the image area to be defined, measured in units of tiles and measured with respect to the top left-hand corner of the display buffer. If **PACKED_FLG** is .TRUE., then **IMG_Y** has to be 0
FRM_X	The x co-ordinate of the first tile in framestore to receive the pixels from the image area, measured in units of tiles, and measured with respect to the top-left-hand corner of framestore
FRM_Y	The y co-ordinate of the first tile in framestore to receive the pixels from the image area, measured in units of tiles, and measured with respect to the top-left-hand corner of framestore
DISPLAY_OPT	Acts as an 8-bit mask, specifying which bits of the framestore pixels selected for output are actually output; in 24-bit working, each of the 3 bytes of a pixel is masked by the same **DISPLAY_OPT** mask. Note: **DISPLAY_OPT** is ignored on DAPs fitted with the DPIO graphics board
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 51 in appendix A for details of the meaning of a non-zero error code

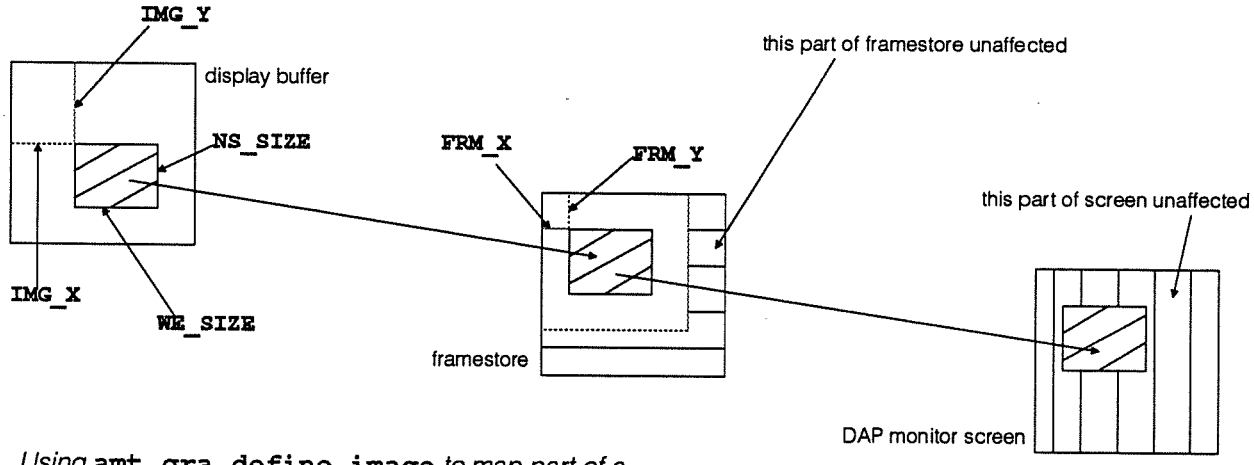
The routine maps a selected area of the display buffer onto part of the framestore. You can map the area onto any part of the framestore, provided that the mapped area does not overlap the edges of the framestore.

The routine also lets you specify that when pixels are sent to framestore, copying does not start at bit 0 of the framestore's

pixels and that only some of the bits in the pixels as received in the framestore are to be output.

If the mapping that **amt_gra_define_image** specifes does not imply a 'shift' of image – that is if **FRM_X=IMG_X**, and **FRM_Y=IMG_Y** – then (subject to the effect of **OFFSET_B_OUT** mentioned below) the whole of the contents of the display buffer over-writes framestore in subsequent calls to **amt_gra_put_frame** or **amt_gra_start_sequence**, even though not all the framestore is output to screen. If however a 'shift' is specified in **amt_gra_define_image**, then only that part of framestore overlapped by the shifted display buffer is over-written.

The diagram below illustrates some of the capabilities of **amt_gra_define_image**. You can map only part of the display buffer onto framestore, so only outputting that part to screen, leaving the rest of the screen image – and some of the rest of the contents of framestore – unaffected when subsequently you call **amt_gra_put_frame** or **amt_gra_start_sequence**.



Using **amt_gra_define_image** *to map part of a display buffer onto framestore and the DAP monitor screen*

The diagram does not show the routine's ability to map the display buffer image onto only some bits of the framestore pixels, or to output to only some bits of the screen pixels.

Bits within a pixel are numbered 0 to *pxl*–1, bit 0 being the most significant, where *pxl* is the number of bits in a pixel. In a framestore pixel *pxl* is 8 for VO–8 and DPIO couplers, 24 for VO–24 couplers.

Normally bit *n* of the selected pixels in the display buffer over-writes bit *n* of the corresponding pixels in framestore. However, if you specify a **OFFSET_B_OUT** greater than 0, then bits 0 to (**OFFSET_B_OUT**–1) in framestore are

unaffected when you subsequently call **amt_gra_put_frame** or **amt_gra_start_sequence**, and bit *n* of the display buffer overwrites bit (*n*+**OFFSET_B_OUT**) of framestore. Here *n* goes from 0 to the size in bits of the pixels in the display buffer.

uses for amt_gra_define_image

One of the limitations on the output of fast-moving images on the DAP is that it takes 20 milliseconds to output a full 8-bit 1024 by 1024 pixel image to screen – and 60 milliseconds for a full 24-bit image. The screen refresh rate is 60 pictures a second so it is not possible to change the whole of the picture every 1/60 of a second (16.7 milliseconds).

.. higher-speed pictures

amt_gra_define_image lets you define an image smaller than 1024 by 1024, so that you can change only part of the image on screen. If you define your image as, say 512 by 512 pixels, then the transfer-to-screen time for such an 8-bit image is 5 milliseconds, and 15 milliseconds for such a 24-bit image. Note that this faster-changing picture on the screen only takes up part of the screen area; there is currently no magnify facility available to 'blow up' part of framestore onto the whole of screen.

.. montage of small images

Another use for the routine is to build up a montage of small images in the framestore and on the screen – by setting **PACKED_FLG** on, declaring the display buffer to be the same size as the small images, and moving the defined image around the framestore to build up the montage. Note, though, that you have no control over screen refresh, and you may get strange visual glitches as each new part of the montage is added to the screen.

.. multi-images

A third use is have two or more full-size images on the screen at the same time, each image associated with different bits in the pixel. You can then set **DISPLAY_OPT** so that only the appropriate bits in the displayed pixels are updated.

If you set **DISPLAY_OPT** and **OFFSET_B_OUT** appropriately, you can, for example, write a single-bit-plane image to any pixel plane in framestore, and display a screen picture in which only the image associated with that bit plane is changed.

2.6 amt_gra_get_lut

amt_gra_get_lut copies the currently-loaded look-up table from the graphics hardware into a selected area of your array store.

Typical calling sequence:

```
INTEGER LUT(256,3),IERR
:
CALL AMT_GRA_GET_LUT(LUT,IERR)
```

The routine takes the arguments:

Argument	*Description*
LUT	Specifies the area of store into which the current look-up table is to be copied
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 49 in appendix A for details of the meaning of a non-zero error code

The routine copies the values held in the hardware look-up table into the array **LUT**. In the current **gralib** implementation the red entries will be in **LUT(,1)**, the green in **LUT(,2)** and the blue in **LUT(,3)**.

You can use this routine together with **amt_gra_put_lut** to create and load back a modified version of the current look-up table. The routine is also useful if an application needs to preserve and re-instate the colour context.

2.7 amt_gra_init_font

amt_gra_init_font associates a font identifier with a selected font. Subsequent calls to **amt_gra_put_characters** can then use that identified font.

Typical calling sequence:

```
INTEGER IFONT,MAX_CELL_X,MAX_CELL_Y,IERR
:
CALL AMT_GRA_INIT_FONT_fontname (IFONT,MAX_CELL_X,MAX_CELL_Y,IERR)
```

The routine takes the arguments:

Argument	*Description*
fontname	The name of the required font. The table on the next page lists the names of the available fonts
IFONT	The variable that on exit from the routine returns the font identifier
MAX_CELL_X	The variable that on exit from the routine returns the width, in pixels, of the largest character cell in the font
MAX_CELL_Y	The variable that on exit from the routine returns the height, in pixels, of the largest character cell in the font
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See appendix A starting on page 49 for details of the meaning of a non-zero error code

Associated with each character in each font is a *character cell*, a notional box that encloses the character, with a suitable amount of space on all sides. The character cells for the various characters in a font will be of the same height, but in general will be of different widths.

You can initialise as many of the available fonts as you want, each font taking about 130 planes of your program block (on a DAP 500 or a DAP 600). Once initialised (with identifiers, perhaps, of **IFONT_1** for **Type_1**, **IFONT_1b** for **Type_1b**, and so on – see below for a list of the available fonts) the initialised fonts are available for calls to **amt_gra_put_characters**.

If you do not call the **amt_gra_init_font** routine at all, then only the default font type (corresponding to an **IFONT** value of 0) is available

The available fonts are:

Name	*Description*	*Maximum size of character cell (width by height)*
Type_1	non-proportional, serif	13 by 24 pixels
Type_1b	non-proportional, serif, bold	14 by 24 pixels
Type_2	non-proportional, serif	12 by 15 pixels
Print_1	proportional, serif	11 by 18 pixels
Print_1b	proportional, serif, bold	11 by 19 pixels
Print_2	proportional, sans serif, large	30 by 32 pixels
default font	non-proportional, serif, small	9 by 12 pixels

The default font is the font that is available with a font identifier of 0.

2.8 amt_gra_init_graphics

amt_gra_init_graphics requests access to the graphics hardware, nominates and describes the first display buffer and sets up the initial drawing parameters to suit this buffer.

Typical calling sequence:

```
INTEGER*n IMAGE(*1024,*1024)
:
INTEGER IERR,MONITOR,OPTION
:
CALL AMT_GRA_INIT_GRAPHICS(MONITOR,IMAGE,OPTION,IERR)
```

The routine takes the arguments:

Argument	*Description*
MONITOR	Specifies which display monitor is to be used. The **MONITOR** parameter will in a future release be used to select an appropriate video output coupler (and associated monitor) from those that may be installed in one or more of the four FIO hardware slots in the DAP.

In the meantime you should set **MONITOR** to **-1** which will connect you to the first available monitor. In the previous release of **gralib**, you were recommended to use a value of **1** for **MONITOR**, which is also currently acceptable, and also connects you to the first available monitor

IMAGE Declares the area of store to be used as display buffer. Normally (as in the above calling sequence) the buffer will be declared as a *1024,*1024 area, but you can declare a display buffer *smaller* than full-screen size. Note, though, if you do declare such a buffer, then unless you use **amt_gra_define_image** (see page 13) to change the image to be displayed to match the smaller buffer, you may get strange results

caution

OPTION Selects between possible modes of operation: defines the colour mode and colour precision as follows:

Value in **OPTION**	*Mode*	*Channels*	*Bits per channel*	*Pixel length*
n	MAPPED	1	*n*=1, 2, 4 or 8	*n*
(16+*n*)	DIRECT	3	*n*=1, 2, 4 or 8	(3**n*)

IERR The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 49 in appendix A for details of the meaning of a non-zero error code

You cannot drive an 8-bit video output coupler in 24-bit mode and any request to do so will fail. However, you can initialise a 24-bit video output coupler for either 8-bit or 24-bit mode.

amt_gra_init_graphics loads a grey scale palette into the colour look-up table, whatever the value of **OPTION**: equal values are loaded into corresponding entries in each of the red, green and blue sub-tables in the table (0 for the 3 entry 1s, .. *n* for the 3 entry (*n*+1)s, ... 255 for the 3 entry 256s).

The contents of your nominated display buffer (**IMAGE** in the above example) are not transferred to the hardware framestore until either routine **amt_gra_put_frame** or routine **amt_gra_start_sequence** is subsequently called.

2.9 amt_gra_magnify

amt_gra_magnify magnifies an input image; the magnification factors in the x and y directions can be different, but both must be powers of 2. The result fills the whole of the output image buffer.

A typical calling sequence:

```
INTEGER IERR
:
INTEGER*n INPUT(*I_X_SIZE,*I_Y_SIZE),DESTINATION(*D_X_SIZE,*D_Y_SIZE)
:
CALL AMT_GRA_MAGNIFY(INPUT,DESTINATION,IERR)
```

The routine takes the following arguments:

Argument	*Description*
INPUT	The image to be magnified. **I_X_SIZE** is the horizontal size of the image, is measured in pixels, and has to be an integral multiple of *ES*, the target DAP edge-size. **I_Y_SIZE** is the vertical size of the image, is measured in pixels, and has to be an integral multiple of *ES*, the target DAP edge-size
DESTINATION	The image buffer to receive the magnified image. **D_X_SIZE** is the horizontal size of the buffer, is measured in pixels, and has to be 2^n x **I_X_SIZE**, where *n* is an integer value. **D_Y_SIZE** is the vertical size of the buffer, is measured in pixels, and has to be 2^m x **I_Y_SIZE**, where *m* is an integer value
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 51 in appendix A for details of the meaning of a non-zero error code

The magnification ratios in the x and y directions are set by the ratios of the declared sizes of the input and output images. Provided the ratios are both powers of 2, they need not be the same; they are:

$$\text{x magnification} = \frac{\texttt{D\_X\_SIZE}}{\texttt{I\_X\_SIZE}}$$

$$\text{y magnification} = \frac{\texttt{D\_Y\_SIZE}}{\texttt{I\_Y\_SIZE}}$$

You can extend the usefulness of the routine by using it in conjunction with normal FORTRAN-PLUS enhanced commands – you can extract **INPUT** from a larger image using **get_mat**, and put the magnified **INPUT** into a buffer larger than **DESTINATION** using **set_mat**.

2.10 amt_gra_put_characters

amt_gra_put_characters puts coloured text in one of the initialised fonts into the nominated display buffer.

Typical calling sequence:

```
INTEGER OP,X,Y,COLOUR,NUM_IN_STRING,IFONT,IERR
:
CHARACTER STRING(NOCR)
:
CALL AMT_GRA_PUT_CHARACTERS(X,Y,STRING,NUM_IN_STRING,COLOUR,IFONT,OP,IERR)
```

The routine takes the arguments:

Argument	*Description*
X	The x co-ordinate of the top-left-hand corner of the start of the displayed character string
Y	The y co-ordinate of the top-left-hand corner of the start of the displayed character string
STRING	The character string to be sent to the display buffer
NUM_IN_STRING	The number of characters to be displayed; only the first **NUM_IN_STRING** characters from **STRING** will be displayed. **NUM_IN_STRING** should not be larger than **NOCR**, the number of characters reserved for **STRING**
COLOUR	The pixel value of the required display of **STRING**
IFONT	The identifier for the font to be used to display **STRING**; the identifier will have been returned by an earlier call to **amt_gra_init_font**. The identifier for the default font is 0
OP	The option to be used to combine the character string pixels with the existing pixels in the display buffer. See below for details
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

All coordinates should be in the range 0 to 1023; if part of a character is outside the range, then the character will be clipped to the edge of the screen. Only the least significant *pxl* bits of **COLOUR** are used, all others are ignored.

Parameter **OP** is used much as **OP** in routine **amt_gra_rasterop** (see page 29) and can take one of two values:

Value of **OP**	*Effect*
1	**Replace** – Each pixel making up the character cells containing the characters being written to the buffer over-writes the relevant existing pixel in the display buffer
2	**XOR** – Each pixel making up the character cells containing the characters being written to the buffer is combined with the relevant existing pixel in the display buffer via a bit-by-bit exclusive-or operation on the two pixel values

amt_gra_put_characters creates a box for each character it writes to the screen, the background colour of the box having a pixel value of 0, and the character shape a pixel value of **COLOUR**. If several characters are being written to the screen, their boxes are joined up to form a rectangular strip holding the required text string. The strip-plus-string is combined with the existing image in the screen buffer in a way defined by the **OP** parameter.

how to get text with no apparent background strip

To write text of pixel value A, having a 'transparent' background strip, over a region with existing pixel value B you would use an **OP** of 2 in your call to

amt_gra_put_characters. You need to work out the **COLOUR** to be supplied on the call; it is the exclusive-OR of A and B. (Since the XOR operation is reversible, if **COLOUR** is XOR (A,B) then XOR (**COLOUR**,B) produces A once more.)

an example, with ILUT of 4

For example, suppose you are using mapped colour and you have set the active colour look-up table by running **amt_gra_set_lut** with an **ILUT** of 4 (see section 2.20 on page 33 for details), and that the existing image in the display buffer has a uniform pixel value of 1, producing a plain red background; under this look-up table, a pixel value of 0 produced the colour black.

Suppose that you want to put white characters on the screen, needing a pixel value of 7 with the **ILUT**=4 look-up table. If you use an **OP** of 1 in **amt_gra_put_characters**, that is replacing the existing image with the characters and their strip, the result on the screen would be a plain red background, into which is cut a black strip containing your white text string. If you specify an **OP** of 2, that is XOR-ing on a bit-by-bit basis the pixel values of the existing image with those of the characters and their strip, the result on the screen would be cyan letters (with a pixel value of 6, the XOR of 7 with 1) on a red background, with no black strip surrounding the letters. To get white letters on a plain red background with no black strip, you would need to specify a **COLOUR** of 6 (cyan), and an **OP** of 2.

2.11 amt_gra_put_dots

amt_gra_put_dots generates a scatter plot of individually-coloured single pixel dots in your nominated display buffer.

Typical calling sequence:

```
INTEGER X(*NOD),Y(*NOD),COLOUR(*NOD),IERR
:
LOGICAL MASK(*NOD)
:
CALL AMT_GRA_PUT_DOTS(X,Y,MASK,COLOUR,IERR)
```

The routine takes the arguments:

Argument	*Description*
X	The variable holding the x co-ordinates of the required dots
Y	The variable holding the y co-ordinates of the required dots
MASK	The logical variable defining which dots are to be added to the display buffer; dots in locations corresponding to .TRUE.s in **MASK** are added
COLOUR	The variable holding the pixel values of the required dots

IERR The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

NOD, the number of dots to be drawn, can take any positive integer value. The number of dots that you can draw in one call to the subroutine is only limited by possible array memory limitations caused by the size of vectors you have to declare for **X**, **Y**, **COLOUR**, and **MASK**.

All co-ordinates should be in the range 0 to 1023; if either the x or y co-ordinate is out of range, the dot will not be drawn. Only the least significant *pxl* bits of **COLOUR** are used, all others are ignored (*pxl* being 1, 2, 4 or 8 for mapped colour; or 3, 6, 12 or 24 bits for direct colour).

Although it may seem most natural to specify the co-ordinates and so on of the required dots as components in vectors (as above), you could equally well have used matrices to hold the various sets of parameters for the dots.

For example, if you had used the sequence:

```
INTEGER NS(*A,*B),WE(*A,*B),HUE(*A,*B),ERROR_CODE
LOGICAL FILTER(*A,*B)
:
CALL AMT_GRA_PUT_DOTS(NS,WE,FILTER,HUE,ERROR_CODE)
```

You would potentially generate (**A** x **B**) individually-coloured dots, although mask **FILTER** might have stopped some being generated.

You can use **amt_gra_set_colour_regime** (see page 31) to specify how the dots are to be combined with the existing image in the display buffer; the default in the current **gralib** implementation is for the pixels of the dots to replace the existing pixels.

2.12 amt_gra_put_frame

amt_gra_put_frame copies the image in the nominated display buffer to the framestore for output to the DAP monitor screen, and frees the buffer for preparation of the next image.

Typical calling sequence:

```
INTEGER IERR
:
CALL AMT_GRA_PUT_FRAME(IERR)
```

The routine's only argument:

IERR The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

The routine copies the data in your display buffer into the hardware framestore; control does not return from **amt_gra_put_frame** until all the data has been copied.

Data transfer time for a full 1024^2 pixel display buffer is 20 milliseconds for 8-bit pixels and 60 milliseconds for 24-bit pixels. Although the data transfer is independent of screen refresh, the system makes sure that the image is not displayed until the transfer is complete.

This type of output lets you update the picture on the screen at random times.

2.13 amt_gra_put_lines

amt_gra_put_lines draws individually-coloured simple lines in the nominated display buffer.

Typical calling sequence:

```
INTEGER XSTART(*NOL),YSTART(*NOL),XFINISH(*NOL),YFINISH(*NOL)
:
INTEGER COLOUR(*NOL),IERR
:
LOGICAL MASK(*NOL)
:
CALL AMT_GRA_PUT_LINES(XSTART,YSTART,XFINISH,YFINISH,MASK,COLOUR,IERR)
```

The routine takes the arguments:

Argument	*Description*
XSTART	The variable holding the x co-ordinates of one end of the required lines
YSTART	The variable holding the y co-ordinates of one end of the required lines
XFINISH	The variable holding the x co-ordinates of the other end (to **XSTART**, **YSTART**) of the required lines
YFINISH	The variable holding the y co-ordinates of the other end (to **XSTART**, **YSTART**) of the required lines
MASK	The logical variable defining which lines are to be added to the display buffer; lines in locations corresponding to .TRUE.s in **MASK** are added
COLOUR	The variable holding the pixel values of the required lines
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

NOL, the number of lines to be drawn, can take any positive integer value. The number of lines that you can draw in one call to the subroutine is only limited by possible array memory limitations caused by the size of vectors you have to declare for **XSTART**, **XFINISH**, **YSTART**, **YFINISH**, **COLOUR**, and **MASK**.

All co-ordinates should be in the range 0 to 1023 and any line for which either start or end points are out of range will not be drawn and you will get an error report, although execution will continue.

Only the least significant *pxl* bits of **COLOUR** are used, all others are ignored (*pxl* being 1, 2, 4 or 8 for mapped colour; or 3, 6, 12 or 24 bits for direct colour).

Although it may seem most natural to specify the start and end co-ordinates and so on of the required lines as components in vectors (as above), you could equally well have used matrices to hold the various sets of parameters for the lines.

You can use **amt_gra_set_colour_regime** (see page 31) to specify how the lines are to be combined with the existing image in the display buffer; the default in the current **gralib** implementation is for the pixels of the rectangles to be **RANK**ed with the existing pixels according to their pixel values.

2.14 amt_gra_put_lut

amt_gra_put_lut loads a colour look-up table from array store into the graphics hardware, giving the advanced user full control over the mapping of colour indices to actual displayed colours, for both mapped and direct colour.

Typical calling sequence:

```
INTEGER LUT(256,3),IERR
:
CALL AMT_GRA_PUT_LUT(LUT,IERR)
```

The routine takes the arguments:

Argument	*Description*
LUT	Specifies the area of store from which a look-up table is to be loaded
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 49 in appendix A for details of the meaning of a non-zero error code

amt_gra_put_lut copies the look-up table entries from the integer array **LUT** into the graphics hardware. The current **gralib** implementation assumes that the entries in the array are **LUT**(*red-values*,**1**), **LUT**(*green-values*,**2**), and **LUT**(*blue-values*,**3**). There are 256 elements in each sub-array; the values in any element should lie between 0 and 255, with 0 meaning black and 255 meaning maximum brightness for that colour.

mapped and direct colour

If 8-bit mapped colour is being used, then corresponding elements in the three arrays together specify the colour that maps to the single index used to access the arrays.

For 24-bit direct colour, the three arrays holding the colour brightness information are accessed independently by three separate indices, each array entry providing a translation for a single colour channel.

look-up tables with mapped colour

For 8-bit colour, if the value of a pixel is 0 then the three look-up table entries **(1,1)**, **(1,2)** and **(1,3)** are accessed to find out the actual colour and brightness to be displayed on the screen at that point; pixel value 255 accesses the three entries corresponding to **(256,1)**, **(256,2)** and **(256,3)**.

look-up tables with direct colour

The 24-bit look-up tables similarly links pixel value 0 to the three entries **1,1**; **1,2**, and **1,3**. The mapping of pixel values onto the red, green and blue arrays is implementation-dependent; in the current implementation the least significant 8 bits access the blue array, the middle 8 bits the green array, and the most significant 8 bits the red array.

Hence pixel value 255 corresponds to array entries **1,1**; **1,2**; and **256,3**. Similarly pixel value 265 corresponds to array entries **1,1**; **10,2**; and **1,3**, and pixel value (256^3–1) corresponds to array entries **256,1**; **256,2**; and **256,3**.

for mapped and direct colour

Equal values in element *n* in the red, green and blue arrays of the look-up table produces a colour ranging from black through grey to white on the screen as that value goes from 0 to 255.

Note that for low-precision graphics, which use shorter pixel lengths, the entries indexed are spread evenly across the whole palette. Hence the 16 possible values for 4-bit pixels index the entries (1, 17, 33, ..., 241) in the palette, while 1-bit (bilevel) images use only the first and the 129th entries.

By default, the 8-bit grey scale palette accessed by an **ILUT** of 1 in **amt_gra_set_lut** is loaded when you call **amt_gra_init_graphics**.

2.15 amt_gra_put_rectangles

amt_gra_put_rectangles draws rectangles in the nominated display buffer. The sides of the rectangles are parallel to the co-ordinate axes. Each rectangle is individually specified in terms of position, size and colour by corresponding components in the relevant variables.

Typical calling sequence:

```
INTEGER X1(*NOR),Y1(*NOR),X2(*NOR),Y2(*NOR)
:
INTEGER COLOUR(*NOR),IERR
:
LOGICAL MASK (*NOR)
:
CALL AMT_GRA_PUT_RECTANGLES(X1,Y1,X2,Y2,MASK,COLOUR,IERR)
```

The routine takes the arguments:

Argument	*Description*
X1	The variable holding the x co-ordinates of one corner of the required rectangles
Y1	The variable holding the y co-ordinates of one corner of the required rectangles
X2	The variable holding the x co-ordinates of the opposite corner (opposite to **X1**, **Y1**) of the required rectangles
Y2	The variable holding the y co-ordinates of the opposite corner (opposite to **X1**, **Y1**) of the required rectangles
MASK	The logical variable defining which rectangles are to be added to the display buffer; rectangles in locations corresponding to .TRUE.s in **MASK** are added
COLOUR	The variable holding the pixel values of the required rectangles
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

You can specify either pair of opposite corners of the required rectangles; their order is not significant.

NOR, the number of rectangles to be drawn, can take any positive integer value. The number of rectangles that you can draw in one call to the subroutine is only limited by possible array memory limitations caused by the size of vectors you have to declare for **X1**, **X2**, **Y1**, **Y2**, **COLOUR**, and **MASK**.

The values of **X1**, **Y1**, **X2** and **Y2** should all be in the range 0 to 1023; a rectangle whose centre (co-ordinates (**X1**+**X2**)/2, (**Y1**+**Y2**)/2) is off-screen will not be drawn and you will get an error report, although execution will continue. **amt_gra_put_rectangles** is a special case of **amt_gra_put_wide_lines**, and **amt_gra_put_rectangles**'s error messages (given in appendix A, on page 50) sometimes only make sense in a **amt_gra_put_lines** context.

Although it may seem most natural to specify the co-ordinates of the corners and so on of the required rectangles as components in vectors (as above), you could equally well have used matrices to hold the various sets of parameters for the rectangles.

You can use **amt_gra_set_colour_regime** (see page 31) to specify how the rectangles are to be combined with the existing image in the display buffer; the default in the current **gralib** implementation is for the pixels of the rectangles to replace the existing pixels.

2.16 amt_gra_put_wide_lines

amt_gra_put_wide_lines draws individually-coloured lines in the nominated display buffer. On a high resolution screen the **amt_gra_put_lines** routine may not create a bold enough line. **amt_gra_put_wide_lines** creates a heavier line.

Typical calling sequence:

```
 INTEGER X1(*NOWL),Y1(*NOWL),X2(*NOWL),Y2(*NOWL),W(*NOWL),HWID(*NOWL),
&        VWID(*NOWL),COLOUR(*NOWL),IERR
 :
 LOGICAL MASK(*NOWL)
 :
 CALL AMT_GRA_PUT_WIDE_LINES(X1,Y1,X2,Y2,W,W,MASK,COLOUR,IERR):
 :
 CALL AMT_GRA_PUT_WIDE_LINES(X1,Y1,X2,Y2,HWID,VWID,MASK,COLOUR,IERR)
```

The routine takes the arguments:

Argument	*Description*
X1	The variable holding the x co-ordinates of one end of the required lines
Y2	The variable holding the y co-ordinates of one end of the required lines
X2	The variable holding the x co-ordinates of the other end (to **X1**, **Y2**) of the required lines
Y2	The variable holding the y co-ordinates of the other end (to **X1**, **Y2**) of the required lines
HWID	The horizontal extent of the rectangular brush that is to create the lines
VWID	The vertical extent of the rectangular brush that is to create the lines
W	The horizontal and vertical extents of the rectangular brush that is to create the lines
MASK	The logical variable defining which lines are to be added to the display buffer; lines in locations corresponding to .TRUE.s in **MASK** are added
COLOUR	The variable holding the pixel values of the required lines
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

NOWL, the number of wide lines to be drawn, can take any positive integer value. The number of wide lines that you can draw in one call to the subroutine is only limited by possible array memory limitations caused by the size of vectors you have to declare for **X1**, **X2**, **Y1**, **Y2**, **W**, (or **HWID** and **VWID**), **COLOUR**, and **MASK**.

Wide lines are drawn with a rectangular brush whose centre follows the path between the start and end points.

The routine would generally be called with the same vector of widths, **W**, supplied for both the horizontal and the vertical

brush extents (as in the first call to **amt_gra_put_wide_lines** above). This would yield symmetrical lines drawn with square brushes. Due to the use of the 'smeared square' algorithm, diagonal lines may be as much as $\sqrt{2}$ times as wide as horizontal or vertical lines.

other uses for routine

You can use **amt_gra_put_wide_lines** to draw more general shapes by specifying different horizontal and vertical widths for the rectangular brush. This technique is shown in the second call to **amt_gra_put_wide_lines** in the calling sequence above.

A line with a horizontal width of zero will be drawn with a brush that consists merely of a vertical line, resulting in a parallelogram with one pair of edges vertical. Similarly, you can draw parallelograms with one pair of edges horizontal by setting their vertical widths to zero.

Although it may seem most natural to specify the start and end co-ordinates and so on of the required lines as components in vectors (as above), you could equally well have used matrices to hold the various sets of parameters for the lines.

You can use **amt_gra_set_colour_regime** (see page 31) to specify how the wide lines are to be combined with the existing image in the display buffer; the default in the current **gralib** implementation is for the pixels of the lines to replace the existing pixels.

2.17 amt_gra_rasterop

amt_gra_rasterop copies image data to your nominated display buffer from another part of array store.

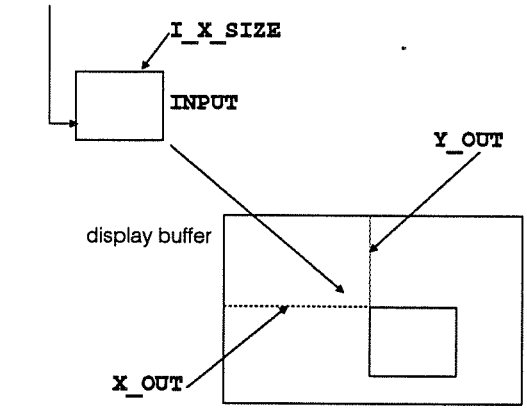
Typical calling sequence:

```
INTEGER*n INPUT(*I-X-SIZE,*I-Y-SIZE)
:
INTEGER OP,IERR,X_OUT,Y_OUT
:
CALL AMT_GRA_RASTEROP(INPUT,X_OUT,Y_OUT,OP,IERR)
```

The routine takes the arguments:

Argument	*Description*
INPUT	The input image.
	I_X_SIZE is the horizontal size of the image to be copied, and is measured in pixels.
	I_Y_SIZE is the vertical size of the image to be copied, and is measured in pixels
X_OUT	The column address, in pixels, of the left edge of the receiving region in the display buffer
Y_OUT	The row address, in pixels, of the top edge of the receiving region in the display buffer
OP	The combination function (see below)

Argument	*Description*
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code



The sketch in the margin illustrates some aspects of what the routine does.

The routine copies an input image (held in a data area in your program block) to a region of the same size in your display buffer.

The input image need not be the same size as the buffer. The location of the input image is in general different from that of the buffer. There is no restriction on the positions of the edges of the input or output regions – that is, they do not have to lie on tile boundaries.

The **OP** parameter specifies how input pixels are combined with pixels in the display buffer. **OP** takes either of the following values:

Value of **OP**	*Effect*
1	**Replace** – Each pixel of the input region over-writes the corresponding pixel of the display buffer region
2	**XOR** – Each pixel of the input region is combined with the corresponding pixel of the display buffer region by a bit-by-bit exclusive-or operation on the two *pxl*-bit pixel values

The top left corner of the output region (specified by **X_OUT**, **Y_OUT**) should be on the screen, but the other corners of the region need not be, and if they are not the resultant image in the display buffer will be clipped accordingly.

caution

You can use the display buffer itself as the input image, but if you do, you should make sure that the input and output regions do not overlap – if they do, the effect is undefined.

2.18 amt_gra_RGB_val and amt_gra_RGB_vals

These colour construction functions make the construction of 24-bit red, green and blue (RGB) colour values easier. **amt_gra_RGB_val** generates a single colour value, while **amt_gra_RGB_vals** returns a vector or matrix of colour values.

A typical use of these colour construction functions:

```
      INTEGER R,G,B, RVM(*300,*300),GVM(*300,*300),BVM(*300,*300),
     &                                X(*300,*300),Y(*300,*300),IERR
      INTEGER*3 COLOUR
      INTEGER*4 COLMAT(*300,*300)
      :
```

```
LOGICAL MASK(*300,*300)
EXTERNAL INTEGER*4 SCALAR FUNCTION AMT_GRA_RGB_VAL
EXTERNAL INTEGER*4 VECTOR FUNCTION AMT_GRA_RGB_VALS
DIMENSION AMT_GRA_LIB_RGB_VALS(*,*)
:
COLOUR = AMT_GRA_RGB_VAL(R,G,B)
COLMAT = AMT_GRA_RGB_VALS(RVM,GVM,BVM)
:
CALL AMT_GRA_CLEAR_SCREEN(RGB_VAL(R,G,B),IERR)
CALL AMT_GRA_PUT_DOTS(X,Y,MASK,COLMAT,IERR)
```

The functions take the arguments:

Argument	*Description*
R	The brightness of the red component in the pixel value being constructed; only the bottom 8 bits of **R** are used to construct the value
G	The brightness of the green component in the pixel value being constructed; only the bottom 8 bits of **G** are used to construct the value
B	The brightness of the blue component in the pixel value being constructed; only the bottom 8 bits of **B** are used to construct the value
RVM	The brightness of the red components in the set of pixel values being constructed; only the bottom 8 bits of each component of **RVM** are used to construct the value
GVM	The brightness of the green components in the set of pixel values being constructed; only the bottom 8 bits of each component of **GVM** are used to construct the value
BVM	The brightness of the blue components in the set of pixel values being constructed; only the bottom 8 bits of each component of **BVM** are used to construct the value

which would clear the screen to a colour whose primaries were specified in **R**, **G** and **B**, and would then put 9,000 different-colour dots on the screen.

'vals either a matrix or vector function

Note that **amt_gra_RGB_vals** can be either a matrix function or a vector function.

These **RGB_val(s)** functions can be used whenever a colour (or set of colours) is required.

You can use them as arguments to **gralib** routines such as **amt_gra_clear_screen**, **amt_gra_put_dots** or **amt_gra_wide_lines**.

Again, you can use the FORTRAN-PLUS command **set_mat** and **gralib**'s **amt_gra_RGB_val** to assign different colours directly to different areas of a 24 bit display buffer.

2.19 amt_gra_set_colour_regime

amt_gra_set_colour_regime lets you choose how pixels are to be combined when new graphics object(s) are added to an existing image in the nominated display buffer. In

the simplest case the existing pixels are overwritten, but other options are available.

Typical calling sequence:

```
INTEGER OP,IERR
:
CALL AMT_GRA_SET_COLOUR_REGIME(OP,IERR)
```

The routine takes the arguments:

Argument	*Description*
OP	Defines how pixels from new object(s) are to be combined with the existing image in the display buffer; takes a value in the range 1 to 7:

Value of **OP**	*Colouring option*
1	**Replace** – New values over-write the existing values in each relevant pixel
2	**XOR** – A logical exclusive-OR operation is carried out between the bit patterns of the new and existing pixel values to derive the final values for each pixel
3	**Rank** – For mapped colour only: the new and existing values held in each pixel are compared and the higher of the two values is used. If you try to use this option in direct colour, you will get an **IERR** of –171 (SetColourRegime: Bad Rule for Mapped Colour)
4	**Add** – The new and existing pixel values are added. The resultant will wrap around if the sum exceeds the maximum possible. This option was provided for mapped colour (where the maximum value is 255), but will work in direct colour. In direct colour, where the 24-bit composite pixel values are added, you may get strange effects if the sum of any pair of the separate colour components exceeds 255. For example: in the current **gralib** implementation the red 8-bit component is the most significant in the 24-bit value, followed by the green, then the blue. If you **Add** to a colour of white (255 for each of the red, green and blue components) a colour of very dark blue (with colour components of 0, 0 and 1 for red, green and blue), then the resultant colour is black – because the resultant 24-bit pixel value is 0! In mapped colour, for example: 10 (old) + 255 (new) = 9.
5	**Sub** – The new pixel value is subtracted from the existing one. The resultant will wrap around if the difference is less than 0. As with **Add**, this option was provided for mapped colour, but will work in direct colour, possibly producing strange results, as suggested above. In mapped colour, for example: 5 (old) – 6 (new) = 255
6	**RGBadd** – For direct colour only. Here each colour channel is added separately, and any excessive values are *clamped* to 255 in order to prevent overflow between channels or wrap around. For example, for 24 bit direct colour the RED channel might be added as 10 (old) + 253 (new) = 255
7	**RGBsub** – For direct colour only. As for **RGBadd**, but results are clamped to a minimum of 0 for each colour component

IERR The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 51 in appendix A for details of the meaning of a non-zero error code

The **Rank** option is designed for use with a colour look-up table that maps darker background colours to lower pixel values, and brighter foreground colours to higher values. By encoding the intended precedence (maybe of 3-D depth) into the pixel values, overlapping objects will maintain the required visibility, regardless of the order in which they are drawn.

Note that some **gralib** routines let you specify a colouring operation explicitly, which then temporarily overrides (but does not change) the current regime.

The default colour regime is not constant for all routines. **Replace** is used for **amt_gra_put_dots**, **amt_gra_put_rectangles**, and **amt_gra_put_wide_lines**, but **Rank** is used for **amt_gra_put_lines**. You should call **amt_gra_set_colour_regime** explicitly rather than relying on the default regimes, since these may change in future releases of **gralib**.

2.20 amt_gra_set_lut

amt_gra_set_lut loads one of the four AMT-supplied standard colour look-up tables into the graphics hardware.

Typical calling sequence for the **amt_gra_set_lut** routine is:

```
INTEGER ILUT(256,3), IERR
:
CALL AMT_GRA_SET_LUT(ILUT,IERR)
```

The routine takes the arguments:

Argument	*Description*
ILUT	Specifies which of the four AMT-supplied standard palettes are to be loaded. See below for details of the different palettes
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 49 in appendix A for details of the meaning of a non-zero error code

It is difficult to calculate the red, green and blue component values that will generate a particular colour, and tedious to specify 256 of them. Where colour is being used to discriminate between different displayed objects, or as a means of displaying a range of data values, it is often enough to select one of **gralib**'s preset colour palettes (that is, colour look-up tables) using the **amt_gra_set_lut** routine.

The palettes are of most use in mapped colour mode, and are described below in terms of mapped colour pixel values:

Value of **ILUT**	*Palette produced*
1	A grey scale – pixel value 0 gives black, 255 gives white (maximum brightness)
2	'Glowing coals' – a colour scale starting at black (pixel value 0), moving through red, orange and yellow, and ending at white (pixel value 255). This palette is intended as an alternative to the grey scale of **ILUT**=1
3	A rainbow – a colour scale starting at red (pixel value 0) and running through a range of rainbow colours – red, orange, yellow, green – to blue (pixel value 255)
4	A repeated colour scale – the 8 colours black, red, green, blue, yellow, magenta, cyan, white are repeated 32 times to provide the complete 256-colour palette. Hence pixel value (0 modulo 8) gives black, (1 modulo 8) gives red, ... (10 modulo 8) gives green, and so on. Note that this palette is unsuitable for shorter pixel lengths (1, 2 or 4 bits) since all displayed colours will be the same

Note that the routines that manipulate the colour look-up table are not generallay useful in direct colour mode. However, as noted in chapter 1, direct colour 24-bit pixel values are also 'translated' by the look-up table.

uses for look-up table with direct colour

One use for this 24-bit translation is to re-allocate the 24 colour bits, using less than 24 bits to hold colour information, and using the 'spare' bit(s) to hold other information – perhaps a flag to say if the pixel is part of a character, or if the pixel is to change colour with some condition, which would be defined elsewhere in the code.

A more esoteric use modifies the colours shown on the monitor screen to compensate for non-linear performance of the monitor's cathode ray tube or its screen phosphors, or of the amplifiers feeding the tube. This technique is called *gamma correction.*

2.21 amt_gra_start_sequence

amt_gra_start_sequence initiates repeated copying to framestore of the nominated display buffer, so that changes can be continuously reviewed.

Typical calling sequence:

```
INTEGER FREQUENCY,IERR
:
CALL AMT_GRA_START_SEQUENCE(FREQUENCY,IERR)
```

The routine takes the arguments:

Argument	*Description*
FREQUENCY	Specifies the time interval, in units of 1/60 of a second, between successive outputs of an image
IERR	The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

At intervals of **FREQUENCY**/60 seconds, the copying process is automatically repeated, until routine **amt_gra_stop_sequence** is called (see section 2.23 on page 36 for details).

uses of the routine

You will find this routine of use when you want to update the picture on the screen at regular intervals. The screen is refreshed from framestore 60 times a second; a value of 20 for **FREQUENCY** would lead to a screen display of 3 different pictures a second. **FREQUENCY** can take any value from 1 to $(2^{31} - 1)$, but for 8-bit colour precision and a 1024 by 1024 pixel displayed image a value of 1 gives the same effect as 2, since the time needed to transfer the 1 Mbyte of your image to framestore (20 milliseconds) is greater than the time to refresh the screen (1/60 seconds, or 16.67 milliseconds).

For 24 bit colour precision, the minimum achievable **FREQUENCY** for a full-screen displayed image is 4, giving 15 different pictures per second, since the transfer time is increased to 60 milliseconds.

If you declare a display buffer of less than 8 or 24-bit precision, or define a displayed image smaller than 1024 by 1024, you can output more images per second. See **amt_gra_define_image** on page 13 for more details.

There is no synchronisation between any subsequent changes you make to the contents of your display buffer and the regular system-controlled data copying from the buffer to framestore. As a result strange effects may appear on the screen if you are writing to part of the display buffer at the same time as that part of the buffer is being copied to framestore.

2.22 amt_gra_stop_graphics

amt_gra_stop_graphics stops video output and releases hardware resources.

Typical calling sequence:

```
INTEGER IERR
:
CALL AMT_GRA_STOP_GRAPHICS(IERR)
```

The routine's one argument:

IERR The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 49 in appendix A for details of the meaning of a non-zero error code

2.23 amt_gra_stop_sequence

amt_gra_stop_sequence stops the repeated display-buffer-to-framestore copying that was previously initiated by a call to **amt_gra_start_sequence**.

Typical calling sequence:

```
INTEGER IERR
:
CALL AMT_GRA_STOP_SEQUENCE(IERR)
```

The routine's one argument:

IERR The error indicator, which is set on exit from the routine; 0 implies successful completion of the routine. See page 50 in appendix A for details of the meaning of a non-zero error code

The system finishes any buffer-to-framestore copying it was carrying out whan the call was issued; the final image transferred remains in the framestore – and therefore on the screen – on exit from the routine.

2.24 amt_gra_turn_off_error_messages

amt_gra_turn_off_error_messages turns off the outputting of messages relating to errors detected by the graphics routines. By default the messages are output.

Typical calling sequence:

```
CALL AMT_GRA_TURN_OFF_ERROR_MESSAGES
```

See **amt_gra_turn_on_error_messages** below for details of the format of the error messages.

2.25 amt_gra_turn_on_error_messages

By default, errors detected by graphics routines cause a diagnostic report to be output. If you have previously turned off this report by a call to **amt_gra_turn_off_error_messages**, then calling **amt_gra_turn_on_error_messages** turns the report on again.

Typical calling sequence:

```
CALL AMT_GRA_TURN_ON_ERROR_MESSAGES
```

Currently the diagnostic report messages are output using the standard FORTRAN-PLUS **Trace** facility. The form of the display depends on the value of the **psam** environment variable **Pattern_mode** (for more details, see [1] or [2], the version of *DAP Series: Program Development* relevant to your host).

For example, if the default value of **Pattern_mode** (**TRUE**) is current, the message:

```
Too many errors
```

might produce the output:

```
FORTRAN-PLUS Trace
FORTRAN-PLUS Subroutine TEST at Line 4 in File myprog.df

Character Scalar Static Data MESS in 8 bits
           dimensions: (1:15)
(1:15)   Too many errors
End of Report
```

If **Pattern_mode** is set to **FALSE**, then the same **TRACE** output would appear as:

```
FORTRAN-PLUS Trace
FORTRAN-PLUS Subroutine TEST at Line 4 in File myprog.df

Character Scalar Static Data MESS in 8 bits
           dimensions: (1:15)
( 1: 3)  T, o (*2)
( 4:12)   , m, a, n, y,  , e, r (*2)
(13:15)  o, r, s
End of Report
```

Chapter 3

Examples

The first example in this chapter is a fragment of DAP code; examples 2 and 3 show complete programs that will run on a DAP connected to a Sun or VAX host.

3.1 Example 1

This example program fragment shows the type of graphics calls required to set up a regular series of frames.

```
entry subroutine graphics
common/pic/picture(*1024,*1024)
integer*1 picture
integer ierr,monitor,frequency
data monitor/1/,frequency/1/
call amt_gra_init_graphics(monitor,picture,8,ierr)
call amt_gra_clear_screen(0,ierr)
call amt_gra_start_sequence(frequency,ierr)
call compute_and_display_picture
call amt_gra_stop_sequence(ierr)
call amt_gra_stop_graphics(ierr)
return
end
```

This fragment could form part of a program: the subroutine **compute_and_display_picture** would do whatever was necessary for the job in hand.

3.2 Example 2

This example produces a picture with a user-defined look-up table and a single call to **amt_gra_put_frame**.

Note the error checking that is carried out every time a call to a **gralib** routine is issued.

Note also that the yellow 'wedge' produced by the program only reaches half maximum brightness on the screen.

The display buffer (**picture**) is defined as **integer*1**. The maximum positive integer that **picture** can hold is 127; 128 to 255 – the numbers needed to specify a brightness greater than half maximum – are stored as the negative numbers – 128 to –1, in two's complement form, and recognosed by the hardware in that form. You may care to

modify the code so that a wedge going to full brightness is produced. Section 3.4 at the end of this chapter suggests one possible way to modify the code.

DAP program

```
C
      entry subroutine graphics
C
C Set up space for image
C
      common/pic/picture(*1024,*1024)
      integer*1 picture
C
C Set up space for a look-up table and a working vector
C
      common/tables/rgb(256,3)
      integer rgb
      integer ierr,monitor
      integer vindex(*1024)
      data monitor/-1/
C
C Create a look-up table to produce shades of yellow
C

      do 1 i=1,256
      rgb(i,1)=i
      rgb(i,2)=i
      rgb(i,3)=0
 1    continue
C
C Start graphics
C
      call amt_gra_init_graphics(monitor,picture,8,ierr)
C
C Check to see if an error has occurred
C
      if(ierr.lt.0)pause 1
C
C Load up the look-up table
C
      call amt_gra_put_lut(rgb,ierr)
C
C Check to see if an error has occurred
C
      if(ierr.lt.0)pause 2
C
C Compute 'test wedge' picture, going from black at left
C to max brightness of 1/2 full yellow at right
C
      call index_vec(vindex)
      vindex = vindex - 1
      picture = matr(vindex/8,1024)
C
```

```
C Now put out the picture to the screen
C
      call amt_gra_put_frame(ierr)
C
C
C Check to see if an error has occurred
C
      if(ierr.lt.0)pause 3
C
C Pause program execution - to allow the
C user to see the picture!
C
      pause 4
C
C When ready, the user has to type 'q'
C to exit psam and exit the program.
C
C Now switch off
C
      call amt_gra_stop_graphics(ierr)
      if(ierr.lt.0)pause 5
      return
      end
```

You only need a skeleton host program to run the DAP program; a suitable one is shown below. The program assumes that the DAP code is in file **ex2d**.

Host program

```
      program ex2host

      external dapcon
      integer dapcon,dconres

C
C  Connect to the DAP, and load it with the executable file in 'ex2d'.
C

      dconres=dapcon('ex2d')
            if (dconres .ne. 0) pause 1

C
C  Pass control to the DAP entry subroutine 'graphics'.
C

      call dapent('graphics')

C
C  On return from the DAP, release all DAP resources for other users
C

      call daprel
      end
```

3.3 Example 3

Now an example pf a more complex host and DAP program; as with example 2, the DAP part of the program can be run on any model of DAP that is fitted with any type of video board.

Host program

```
C  This is the FORTRAN code for the host part of the program to
C  generate a test card on the DAP monitor screen.
C

      program testgrid

      integer finish

C
C  Initialise the system - that is, set up the parameters for the default
C  test card, then connect to the DAP.
C

      call init

 100  continue

C
C  Plot the test card
C

      call plotcard

C
C  Having displayed the default grid and background, offer the chance
C  to change parameters and display a new test card.
C

      write (*,*) ' '
      write (*,*) 'To change the parameters and re-display, select   0'
      write (*,*) ' '
      write (*,*) 'To exit, select                                   1'
      write (*,*) ' '
      read  (*,*)  finish

      if (finish.eq.1) goto 200

      call getparams

      goto 100

C
C  Pass control to the DAP entry subroutine 'cease', which will close
C  close the DAP down. On return from the DAP release it.
C
```

```
 200  call dapent('cease')
      call daprel

      stop
      end

C-----------------------------------

      subroutine init

C
C  Load the parameters for the initial test card into the COMMON blocks
C  'colours' and 'linesep'.
C
C  Connect to the DAP, load into the DAP the object code in file 'ex3d',
C  and pass control to the DAP at entry subroutine  'init'.
C

      common/colours/gridcol(3),bgcol(3)
      integer gridcol,bgcol

      common/linesep/spacing
      integer spacing

      external dapcon
      integer dconres,dapcon

      gridcol(1)=255
      gridcol(2)=255
      gridcol(3)=255

      bgcol(1)=0
      bgcol(2)=100
      bgcol(3)=0

      spacing=32

      dconres=dapcon('ex3d')
            if (dconres .ne. 0) pause 101

      call dapent('init')

      return
      end

C---------------------------------

      subroutine getparams

      common/colours/gridcol(3),bgcol(3)
      integer  gridcol,bgcol

      common/linesep/spacing
      integer spacing
```

```
      write (*,*) 'First, choose the spacing between grid lines:'
      read  (*,*)  spacing
      write (*,*) ' '


      write (*,*) 'Now choose the colour of the background:'
      write (*,*) ' '
      write (*,*) 'First the red component (0-255):'
      read  (*,*)  bgcol(1)
      write (*,*) 'Next the green component (0-255):'
      read  (*,*)  bgcol(2)
      write (*,*) 'Last the blue component (0-255):'
      read  (*,*)  bgcol(3)
      write (*,*) ' '
      write (*,*) ' '

      write (*,*) 'Lastly, choose the colour of the grid lines: '
      write (*,*) ' '
      write (*,*) 'First the red component (0-255):'
      read  (*,*)  gridcol(1)
      write (*,*) 'Next the green component (0-255):'
      read  (*,*)  gridcol(2)
      write (*,*) 'Last the blue component (0-255):'
      read  (*,*)  gridcol(3)
      write (*,*) ' '
      write (*,*) ' '

      return
      end

C-----------------------------------

      subroutine plotcard
C
C  Pass the test card parameters to the DAP, and pass control to the
C  DAP entry subroutine 'plot'.
C
      common/colours/gridcol(3),bgcol(3)
      integer gridcol,bgcol

      common/linesep/spacing
      integer spacing

      call dapsen('colours',gridcol,6)
      call dapsen('linesep',spacing,1)
      call dapent('plot')

      return
      end
```

DAP program

```
C  This is the FORTRAN-PLUS enhanced code for the DAP part of the program
C  to generate a test card on the DAP monitor screen.
C

      entry subroutine init

      common/screen/picture
      integer*1 picture(*1024,*1024)
      integer error_rep

      call amt_gra_init_graphics(-1,picture,8,error_rep)
           if (error_rep.ne.0) pause 1

      return
      end

C--------------------------------------
C--------------------------------------

      entry subroutine plot

      integer error_rep

      call setscreen

      call drawlines

      call amt_gra_put_frame(error_rep)
           if (error_rep.ne.0) pause 7

      return
      end

C-------------------------------------

      subroutine setscreen

      integer lut(256,3),error_rep

      common/colours/gridcol(3),bgcol(3)
      integer gridcol,bgcol

C
C  Convert the test card colours, in COMMON block 'colours', from host to
C  DAP mode.
C

      call convhtod(gridcol,6)

C
```

```
C
C  Load the required background and grid line colour components into the
C  in scalar array 'lut', and specify 'lut' as the current look-up table.
C

      do 10 i=1,3
            lut(1,i)=bgcol(i)
            lut(2,i)=gridcol(i)
 10   continue

      call amt_gra_put_lut(lut,error_rep)
            if (error_rep.ne.0) pause 2

      call amt_gra_clear_screen(0,error_rep)
            if (error_rep.ne.0) pause 3

      return
      end

C-------------------------------------------------

      subroutine drawlines

      common/linesep/spacing
      integer spacing,error_rep
      integer coords(*1024)

C
C  Convert the test card grid line spacing, in COMMON block 'linesep',
C  from host to DAP mode.
C

      call convhtod(spacing,1)

C
C  Put steps of 1 in the vector 'coords', then change them to steps of
C  the required grid line spacing. Finally, set the last component that
C  will be used in the vector to be equal to 1023 - to generate the
C  right-hand and bottom lines of the grid.
C

      call index_vec(coords)
      coords=(coords-1)*spacing
      coords(1023)=1023

C
C  Call the routines to draw the grid lines: first the vertical lines,
C  then the horizontal lines. The lines are drawn from top to bottom, from
C  left to right.
C
C  vec(0,1024) generates a 1024-component vector, each component being
C  0. Only those lines whose 'coords' co-ordinates are less than 1024
C  are plotted; the colour of each line is specified by 1 in the look-up
C  table.
```

```
      call amt_gra_put_lines(coords,vec(0,1024),coords,vec(1023,1024),
     &                       coords.lt.1024,vec(1,1024),error_rep)
            if (error_rep.ne.0) pause 5

      call amt_gra_put_lines(vec(0,1024),coords,vec(1023,1024),coords,
     &                       coords.lt.1024,vec(1,1024),error_rep)
            if (error_rep.ne.0) pause 6

      return
      end

C--------------------------------
C--------------------------------

      entry subroutine cease

      integer error_rep

      call amt_gra_stop_graphics(error_rep)
            if (error_rep.ne.0) pause 8

      return
      end
```

3.4 Example 2 alternative solution

As mentioned on page 39, you need to use a number in the range –128 to –1 to specify brightness in the range ½ to full. In example 2, only zero to ½ brightness was specified; one way to modify the DAP program in example 2 so as to produce a full-brightness wedge is given below. The changes – a new line and then a changed line – are marked with a vertical line in the margin on the next page.

DAP program

```
C
      entry subroutine graphics
C
C Set up space for image
C
      common/pic/picture(*1024,*1024)
      integer*1 picture
C
C Set up space for a look-up table and a working vector
C
      common/tables/rgb(256,3)
      integer rgb
      integer ierr,monitor
      integer vindex(*1024)
      data monitor/-1/
C
C Create a look-up table to produce shades of yellow
C
```

```
      do 1 i=1,256
      rgb(i,1)=i
      rgb(i,2)=i
      rgb(i,3)=0
 1    continue
C
C Start graphics
C
      call amt_gra_init_graphics(monitor,picture,8,ierr)
C
C Check to see if an error has occurred
C
      if(ierr.lt.0)pause 1
C
C Load up the look-up table
C
      call amt_gra_put_lut(rgb,ierr)
C
C Check to see if an error has occurred
C
      if(ierr.lt.0)pause 2
C
C Compute 'test wedge' picture, going from black at left
C to max brightness of full yellow at right
C
      call index_vec(vindex)
      vindex = vindex - 1
      vindex (vindex .ge. 512) = vindex - 1024
      picture = matr(vindex/4,1024)
C
C Now put out the picture to the screen
C
      call amt_gra_put_frame(ierr)
C
C
C Check to see if an error has occurred
C
      if(ierr.lt.0)pause 3
C
C Pause program execution - to allow the
C user to see the picture!
C
      pause 4
C
C When ready, the user has to type 'q'
C to exit psam and exit the program.
C
C Now switch off
C
      call amt_gra_stop_graphics(ierr)
      if(ierr.lt.0)pause 5
      return
      end
```

Appendix A

Library-defined error messages

Error messages are output if the **gralib** routine **amt_gra_turn_on_error_messages** is active (the default state), and FORTRAN-PLUS **Trace** is turned on (also the default state – see sections 2.24 and 2.25 for more details).

Errors of the type documented here are unusual, but if they do occur then often several will occur at the same time. Sometimes there is a single cause for all the errors, that cause probably being associated with a failure in **amt_gra_init_graphics**.

Possible error messages include:

Generic errors

```
-1              Graphics system not initialised
```

Errors in amt_gra_init_graphics

```
-10             Monitor already in use
-11             Unknown monitor number
-12             Can't open the monitor
```

Errors in amt_gra_stop_graphics

```
-20             Device not open
```

Errors in amt_gra_set_lut

```
-30             The special palette requested is not in the range [1 - 4]
-31             An error has occurred in setting special palette 1
-32             An error has occurred in setting special palette 2
-33             An error has occurred in setting special palette 3
-34             An error has occurred in setting special palette 4
```

Errors in amt_gra_put_lut

```
-40             An error has occurred in setting the palette
```

Errors in amt_gra_get_lut

```
-50             An error has occurred in retrieving the palette
```

Errors in amt_gra_put_frame

-70	An error has occurred in outputting a frame (PUT)
-71	An error has occurred in outputting a frame (SWAP)
-72	An error has occurred in turning the video on

Errors in amt_gra_start_sequence

-80	Invalid FREQUENCY parameter
-81	An error has occurred in starting sequenced frame output

Errors in amt_gra_stop_sequence

-90	An error has occurred in stopping sequenced frame output

Errors in amt_gra_clear_screen

-100	The supplied background colour is out of range [0 - 255]

Errors in amt_gra_put_lines and in amt_gra_put_wide_lines and in amt_gra_put_rectangles

-110	No lines set in mask
-111	Some endpoints out of range
-112	There are no endpoints in range for which mask is set
-113	Some requested lines could not be drawn
-114	No lines could be drawn

Errors in amt_gra_put_dots

-120	Data out of range in X, points not plotted
-121	Data out of range in Y, points not plotted
-122	Data out of range in X and Y, points not plotted

Errors in amt_gra_put_characters

-130	Invalid font number
-131	Font not initialised
-132	Invalid operation code

Errors in amt_gra_rasterop

-140	Rasterop warning: Rasterop area goes beyond input
-141	Rasterop error: Input image size is incorrect
-142	Rasterop error: In width of rasterop area
-143	Rasterop error: In height of rasterop area
-144	Rasterop error: Rasterop area is outside input area
-145	Rasterop error: Rasterop area is outside output area
-146	Rasterop error: Invalid operation code

Errors in amt_gra_copy_image

```
-150            Copyimage error: Negative parameter value
-151            Copyimage error: Parameter out of range
-152            Copyimage error: Parameter not a multiple of DAP size
```

Errors in amt_gra_set_colour_regime

```
-170            SetColourRegime: Rule out of range
-171            SetColourRegime: Bad Rule for Mapped Colour
```

Errors in amt_gra_define_image

```
-160            WE_SIZE out of range
-161            NS_SIZE out of range
-162            BITS_PER_PIXEL out of range
-163            BIT_OFFSET out of range
-164            IMG_X out of range
-165            IMG_Y out of range
-166            FRM_X out of range
-167            FRM_Y out of range
```

Errors in amt_gra_magnify

```
-180            Invalid X magnification
-181            Invalid Y magnification
```

System error code

```
-999            System Error: Unknown error code
```

Appendix B

Specification for routine `magnify`

The routine **magnify** has been added to the version of the **gralib** library issued with release 3.3 DAP basic software.

magnify belongs to the group of **gralib** calls that were available at release 3.2 and earlier, but is not documented in the latest edition of the relevant manual *DAP Series: Low-level Graphics Library* (man017.04).

You might consider including a copy of this sheet in your copy of man017, until such time as the manual is updated.

magnify

The routine lets you magnify part (or all) of an image; the magnification factors in the x and y directions can be different, but both must be positive integer powers of 2. The N–S dimension of the magnified sub-image has to be the same as the N–S dimension of the destination image buffer; the W–E dimension of the destination buffer has to be as least as large as the W–E dimension of the magnified sub-image.

A typical calling sequence:

```
 INTEGER I_WE_SIZE,I_NS_SIZE,I_WE_START,I_NS_START,I_WE_EXT,I_NS_EXT,
&        D_WE_SIZE,D_NS_SIZE,WE_MAG,NS_MAG,PXL,OFFSET
 :
 INTEGER*n INPUT(,,I_NS_SIZE,I_WE_SIZE),DESTINATION(,,D_NS_SIZE,D_WE_SIZE)
 :
 CALL MAGNIFY(INPUT(,,I_NS_START,I_WE_START),DESTINATION,I_WE_EXT,I_NS_EXT,
&        PXL,WE_MAG,NS_MAG,OFFSET)
```

magnify takes the arguments:

Argument	*Description*
INPUT	The image, a part (or all) of which is to be magnified. **I_WE_SIZE** is the horizontal size of the whole of **INPUT**, and is measured in units of tiles. **I_NS_SIZE** is the vertical size of the whole of **INPUT**, and is measured in units of tiles
I_WE_START	The horizontal co-ordinate, measured in units of tiles, of the top-left-hand corner of the part of **INPUT** to be magnified, and is measured with respect to the top-left-hand corner of **INPUT**

Argument	*Description*
I_NS_START	The vertical co-ordinate, measured in units of tiles, of the top-left-hand corner of the part of **INPUT** to be magnified, and is measured with respect to the top-left-hand corner of **INPUT**. The pair **I_WE_START** and **I_NS_START** are optional; if they are not specified, then the whole of **INPUT** is magnified. You have to specify both or none
I_WE_EXT	The horizontal extent, measured in units of tiles, of the part of **INPUT** to be magnified
I_NS_EXT	The vertical extent, measured in units of tiles, of the part of **INPUT** to be magnified
WE_MAG	The required magnification in the horizontal direction; it need not be the same as **NS_MAG**, but must be a positive integer power of 2
NS_MAG	The required magnification in the vertical direction; it need not be the same as **WE_MAG**, but must be a positive integer power of 2
DESTINATION	The image buffer to receive the magnified sub-image **D_WE_SIZE** is the horizontal size of the buffer, and is measured in units of tiles. **D_WE_SIZE** has to be at least (**WE_MAG** x **I_WE_SIZE**). **D_NS_SIZE** is the vertical size of the buffer, and is measured in units of tiles. **D_NS_SIZE** has to be (**NS_MAG** x **I_NS_SIZE**).
PXL	The length in bits of the pixels in **INPUT** and **DESTINATION**
OFFSET	The 'distance', in units of tiles, between the last tile in one column of the part of **INPUT** to be magnified, and the first tile in the next column

For the routine to work without error the magnified sub-image has to be no larger than the display buffer – **(,,32,32)** on a DAP 500, – **(,,16,16)** on a DAP 600

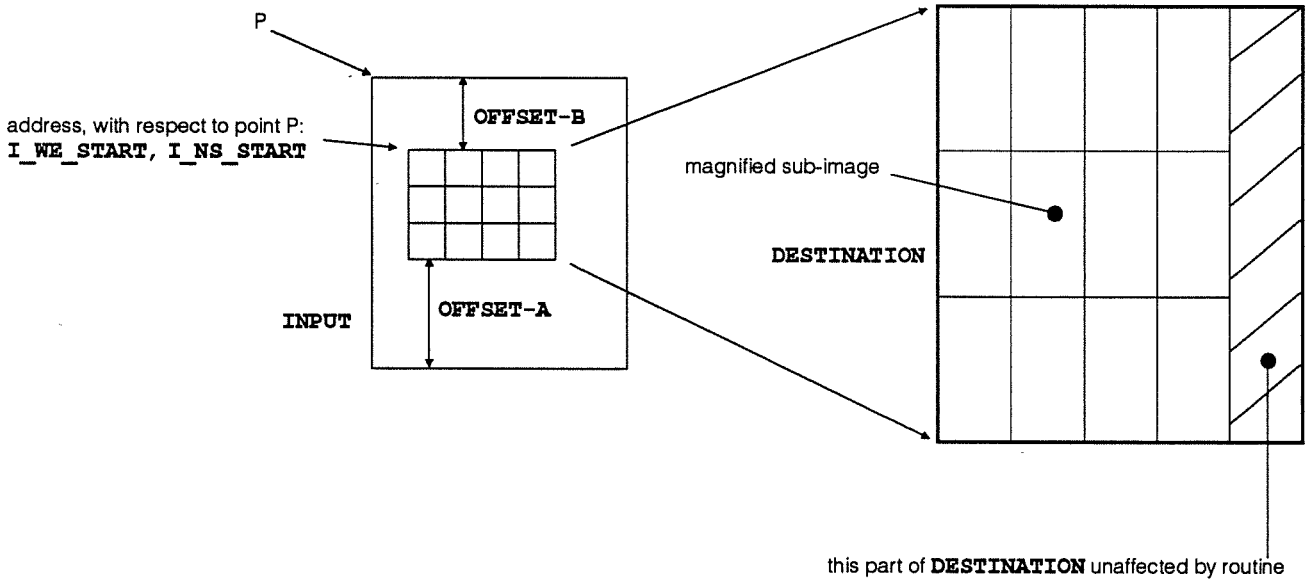
The operation of **magnify** is illustrated in the diagram on the opposite page, and might be the result of the call:

```
INTEGER*1 INPUT(,,8,7),END_PIC(,,12,10)
INTEGER PXL
:
CALL MAGNIFY(INPUT(,,2,1),DESTINATION,4,3,PXL,2,4,5)
```

In the typical calling sequence given on page 53 values for the various parameters were supplied in variables. In the call illustrated here the variables have been replaced by the values:

I_WE_SIZE: 7 (tiles) **I_WE_START**: 1 (tile) **I_WE_EXT**: 4 (tiles) **WE_MAG**: 2
I_NS_SIZE: 8 (tiles) **I_NS_START**: 2 (tiles) **I_NS_EXT**: 3 (tiles) **NS_MAG**: 4
D_WE_SIZE: 10 (tiles) **OFFSET** (= **OFFSET-A** + **OFFSET-B**): 5
D_NS_SIZE: 12 (tiles)

In this case **DESTINATION** is larger than necessary in the W-E direction to receive the magnified part of **INPUT**.


P
address, with respect to point P:
I_WE_START, I_NS_START
OFFSET-B
INPUT
OFFSET-A
magnified sub-image
DESTINATION
this part of DESTINATION unaffected by routine

Index

This index lists all the commands and their associated parameters available to you in **gralib**. All non-alphabetic entries to the index are grouped together under the ! heading immediately below this introduction.

L

M

P

R

S

T

U

W

Reader comment form

AMT
Reading, UK

Any comments you care to make, whether reporting bugs in the manual or making more general comment, about this or any AMT publications will help us improve their quality and usefulness. To report bugs, if you have the time, the ideal way from our point of view is to send us a photo-copy of the relevant page, with the bug marked on it. If you are in the UK, please use our FREEPOST address to send us the copy.

If you also can spare the time to fill in the mini-questionnaire below that would be doubly useful to us. To send us this form, please fold it as indicated, and post it – postage is pre-paid for the UK.

Comments

Title of publication: **Low level graphics library (enhanced)**(man117.01) / other – please specify:

. . . . . .

My name and job title: . . . . . .

My department: . . . . . .

My company: . . . . . .

My company address: . . . . . .

. . . . . .

My telephone number – country: . . . . . number: . . . . . .

I used the publication:

- ☐ As an introduction to the subject
- ☐ To teach myself
- ☐ To teach others
- ☐ As a reference manual
- ☐ Other – please specify

I found the contents:

	True	Partly true	Not true
Helpful	☐	☐	☐
Accurate	☐	☐	☐
Written clearly	☐	☐	☐
Well illustrated	☐	☐	☐
Well indexed	☐	☐	☐
Other – please specify	☐	☐	☐

. . . . . .

Thank you for your help.

23 May 89

Second fold →

Third fold ↓

No postage needed for posting in the UK.
If posting outside UK, please stick stamps to normal value.

← First fold

Third fold ↓

Publications Manager
Active Memory Technology Ltd
FREEPOST (RG 1436)
Reading
Berkshire RG6 1BR
United Kingdom

Fourth fold ↑

Second fold →

Tuck into third fold

Fourth fold ↑

← First fold