An Image Display Interface for Astronomical Image Processing

Editors

David L. Terrett
Rutherford Appleton Laboratory

Peter M. B. Shames
Robert J. Hanisch
Space Telescope Science Institute

Contributors

Rudi Albrecht*	Space Telescope – European Coordinating Facility
Klaus Banse*	European Southern Observatory
Stephen Baron	Space Telescope Science Institute
Eric Greisen	National Radio Astronomy Observatory
Robert Hanisch*	Space Telescope Science Institute
Walter Jaffe	Sterrewacht Leiden
Samy Kijner	Space Telescope Science Institute
Fabio Pasian*	Osservatorio Astronomico di Trieste
Mauro Pucillo*	Osservatorio Astronomico di Trieste
Paolo Santin*	Osservatorio Astronomico di Trieste
Skip Schaller	University of Arizona – Steward Observatory
Giorgio Sedmak	Osservatorio Astronomico di Trieste
Peter Shames*	Space Telescope Science Institute
Cliff Stoll	University of California –Berkeley
David Terrett*	Rutherford Appleton Laboratory
Douglas Tody	National Optical Astronomy Observatories
Jay Travisano	Space Telescope Science Institute
Pat Wallace	Rutherford Appleton Laboratory
Don Wells	National Radio Astronomy Observatory

*Participant in the IDI working group

May 1988

Acknowledgments

Many people have contributed time, energy, thought and support to the design of these interfaces. Most recently, a working group consisting of Rudi Albrecht (ST-ECF), Klaus Banse (ESO), Bob Hanisch (STScI), Fabio Pasian (Trieste), Mauro Pucillo (Trieste), Paolo Santin (Trieste), Peter Shames (STScI), and Dave Terrett (RAL) have held a series of meetings to define the functions and detailed specifications for the interface. The work of this group was aided immensely by the experience of the Trieste and ESO participants in doing preliminary implementations.

The original instigation for developing an image display interface came from Giorgio Sedmak during a European Data Analysis Workshop held in Paris in May 1985. Shortly after this conference a group consisting of Albrecht, Banse, Eric Greisen (NRAO), Walter Jaffe (then at STScI), Shames, Cliff Stoll (then at STScI), and Terrett met to define the basic functional requirements and operational structure for the interface. The result of this meeting was a draft document which described the abstract device model for the interfaces and basically forms the first section of the present document.

Since the initial meeting there have been a number of other working sessions, discussions in open forums, and meetings where the interfaces were discussed and further refined. There are now more than five implementations of earlier versions of the interface, built both in Europe and the US.

During the course of developing the IDI specifications, significant contributions were also made by Steve Baron (STScI), Samy Kijner (STScI), Doug Tody (NOAO), Jay Travisano (STScI), Pat Wallace (RAL), and Don Wells (NRAO). Institutional support and encouragement have been provided by the American Astronomical Society and its Working Group on Astronomical Software (WGAS), Astronet (Italy), ESO, Starlink (UK), the ST-ECF European WGAS, and STScI.

The editors wish to thank S. G. Neff for a critical reading of the manuscript. Although there have been many contributors to the interface, any errors or omissions in this document are the responsibility of the editors.

David L. Terrett
Peter M. B. Shames
Robert J. Hanisch

Abstract

The use made of image display systems in astronomical data reduction is discussed, and a conceptual model of an image display device that supports these uses and is similar to commonly available hardware is developed. This model was used as the basis for designing a set of image display device interface routines. The interface routines support both low- and high-end display devices and a variety of device configurations. The use of high-end imaging workstations is also supported, allow the interface does not attempt to provide a windowing environment. Device configuration choices may be handled via use of one or more standard settings or by use of a negotiated configuration. The interfaces are presented in a language independent fashion with language specific bindings for FORTRAN and C. Several implementations of the early versions of this interface have been built and have demonstrated the practicality of the interface, including the actual port of image display applications software from one device to another.

Contents

1 Introduction

Major astronomical data analysis systems require significant software efforts costing many man-years, and tend to be developed at the central facilities that are responsible for telescopes and instruments. The bulk of astronomical image data is generated at a few major observatories, but these institutions typically cannot provide the computational resources to support full data analysis for all users. Reliance is placed on the ability of users to complete the data analysis at their home institutions, and most of the data analysis systems that have been developed in the last few years are expected to be used at more than one institution. The need for portable software is self-evident.

In order to achieve this portability, much effort has gone into defining portable analysis environments with cleanly defined interfaces to the host operating system and the various devices required for interaction with, and display of, data. The fact that a certain level of commonality has existed at the hardware and operating system levels (*i.e.*,VAXs running the VMS[1] operating system) has helped enormously in this effort, since portability is more easily achieved where common systems are in use. However, recent developments in hardware (*e.g.*, powerful single-user workstations and mini supercomputers) and in operating systems (*e.g.*, the widening use of the UNIX[2] operating system), are showing the weaknesses of this limited approach to software portability.

There are some major challenges in developing truly portable software systems, but even within the restricted environment of a single manufacturer and operating system there are problems to be resolved. Different kinds of terminals, graphics displays, interactive input devices, or image frame buffers and displays will be in use at different sites, and must be easily accommodated by the software. Indeed, this diversity may exist within one site, especially where new or different technologies are used for similar purposes. Technology is offering interesting new devices at a rapid enough rate that the ability to adapt new, more powerful, or more cost effective devices is itself a requirement.

Several major systems now exist which are in use at many sites worldwide and much practical experience in the implementation of applications software for image display devices has been gained. At a meeting of the European Working Group on Astronomical Software, held in Paris in May 1985, the device dependence of most image display software was identified as the most serious stumbling block that remained for many end-user institutions. As a result of this meeting a workshop was held at the Space Telescope Science Institute in June 1985, attended by representatives from several major United States and European observatories. The participants of the workshop first reviewed the capabilities of the various image display devices in use in astronomy and discussed the typical usage of such devices. They then defined a uniform model for an image device interface that would accommodate the range of devices and display techniques that are currently in use and that could support the extensions that would inevitably be required.

Subsequent meetings have refined the model in the light of comments by interested parties and the experience gained by a number of trial implementations on a range of different devices. This document reviews the image display device model used to design the Image Display Interface (IDI), and contains a definition of the interface detailed enough to allow programmers to proceed with its implementation.

[1] VAX and VMS are trademarks of the Digital Equipment Corporation.
[2] UNIX is a trademark of AT&T Bell Laboratories.

It is recognized that the current interface must be extensible so that features available on new image display devices can be accommodated without users having to go outside the interface. Therefore, the set of 'optional' routines defined here is open-ended. The IDI working group will remain active in order to coordinate the addition of new routines to the interface. Users who design new routines or who find it too limiting in certain areas are invited to contact members of the working group with their suggestions. The working group will periodically update the interface description, including new routine descriptions, and will help to distribute information regarding existing implementations.

1.1 The Interface Problem

Various standards efforts promise to supply solutions in some interface problem areas, particularly for terminals, graphics displays, and hardcopy devices, but image displays remain an area of concern because no widely accepted standard now exists. Terminal interfaces can almost be ignored if simple line oriented models of user interaction are chosen, but some concept like Termcap, from UNIX, can usefully be applied if simple editing functions are to be provided on a variety of terminal devices. More extensive terminal interfaces and full screen editing functions can be built with the CURSES package, also from UNIX, that optimizes screen updating based again upon a simple definition of terminal characteristics.

For presentation of line graphics the availability of hardware that emulates the Tektronix 4010 is a help, but much work is usually needed before other graphics devices (such as hardcopy plotters) can be used with an item of foreign software. A Graphcap facility, structured like Termcap, has been successfully used to handle the common variants among Tektronix 4010 emulations, but vastly different devices still require specialized driver software. The emergence of the Graphical Kernel System (GKS) as an international standard promises to solve this problem as soon as cost effective implementations become commonplace. GKS provides a standard set of subroutines for driving graphics devices and also provides the tools necessary for writing programs which adapt themselves to the characteristics of the device being used. This allows well written software to be run on a range of devices with differing capabilities while still fully exploiting the facilities available in the hardware.

The situation for image displays is far worse. There is no equivalent of the Tektronix 4010 protocol, and GKS does not solve the problem. An image construct (called a cell array) is provided in GKS, and a video look-up table can be controlled, but the other capabilities available on all but the most primitive image displays are not addressed. GKS also has an escape mechanism for addressing otherwise inaccessible capabilities, and the IDI Working Group investigated the use of escape functions to control image display devices. It was concluded, however, that the functions needed to interact with the rest of GKS to such an extent that implementation was not feasible with the amount of effort available. Also, access to the source of a GKS implementation would be required and this would preclude the use of commercial implementations.

Although the integration of image displays with GKS or some other international graphics standard remains a long term goal, a more immediate solution is required. The strategy adopted here is to define a set of high level interface routines which will satisfy the needs of the majority of applications programs, and not to attempt to address the more esoteric features of individual models of display. The design of the current interface was governed by the desire to define a virtual image device that mapped into both existing hardware and existing applications code for image processing, yet provided capabilities for taking advantage of device dependent features where available.

Some of the image processing programs in use in astronomy, most notably AIPS[3] and IDL[4], and now STSDAS[5], IRAF[6], and MIDAS[7], have defined sets of routines that isolate the access to the image display device in a small set of interface routines. These routines provide a certain level of portability, but typically permit elements of the underlying hardware, or of the surrounding system, to appear at higher levels of the applications code. In order to promote a higher level of portability of these software systems to different sites, and to allow new devices to be more readily accommodated, a new set of independent interface routines is required. These routines, which are defined in this document, are built upon the virtual device model that is described fully in Section **??**.

1.2 Scope of the Solution

When considering any standard it is important to consider the aims of the standard and the range of hardware and software being addressed.

Hardware. The image display hardware addressed by this interface ranges from simple frame store devices with one image memory and one color look-up table, to devices with multiple image memories, overlay graphics and text memories, multiple color look-up tables, hardware zoom and pan, and a wide range of interactive devices (track ball, joystick, mouse). These displays can often be configured in a variety of ways, with differing numbers and sizes of pixel memories. Unfortunately the rules that govern the possible configurations tend to be closely tied to the details of the hardware implementation, and therefore difficult to generalize.

Pixel arithmetic and image processing functions are not included in the interface because experience shows that, where these facilities exist, they are generally not used. This is probably because when astronomers use some image processing algorithm they are interested in more than just the visual effect on a displayed image. Thus, the operation must be carried out on the original (probably floating point) data, and not on the 8 or 12 bit integer representation of the data stored in the image display memory.

Single-user workstations are beginning to displace the traditional hardware setup based on a multi-user minicomputer, graphics terminal, and image display, but the cost per user is still too high for this to have become the norm. However, the importance of workstations is sure to grow, and therefore we have attempted to ensure that the interface can be implemented on workstations in a way that can take advantage of the new facilities that they offer.

What we have chosen to define in the interface is a set of functions that deals with the manipulation of the various hardware components in an abstract sense. The hardware specific details of how this is done are buried within the interface implementation. Thus,

[3] AIPS, the Astronomical Image Processing System, is developed and distributed by the National Radio Astronomy Observatory, Edgemont Road, Charlottesville, VA 22903.

[4] IDL, the Interactive Data Language, is distributed and licensed by Research Systems, Inc., 2021 Albion Street, Denver, CO 80207.

[5] STSDAS is the Space Telescope Science Data Analysis System, which is developed and distributed by the Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218.

[6] IRAF, the Image Reduction and Analysis Facility, is developed and distributed by the National Optical Astronomy Observatories, P. O. Box 26732, Tucson, AZ 85726.

[7] MIDAS, the Munich Image Data Analysis System, is developed and distributed by the European Southern Observatory, Karl-Schwarzschild Strasse 2, D-8046 Garching bei München, West Germany.

the actions that are important to the user (and thus to the applications programmer) are the ones visible at the interface. These include routines for the selection and display of various memories and overlay data, the modification of color look-up tables and transfer functions, and the dynamic manipulation of various hardware elements.

Software. This interface has been designed to allow astronomical applications programs to be ported between different image display systems and to minimize the amount of effort needed to get existing programs running on a new device. Because they are expected to be in wide use in the community, implementations for most of the more popular devices and systems should be widely available. They are not intended to provide a particularly convenient interface definition for the applications program writer, although in many cases applications programs will indeed call the interface directly. Rather, the interface includes fairly basic functions, and programmers will certainly want to design higher level utilities layered on the interface routines.

Simple line drawing and text facilities are defined, but it is not intended that these should be used for generating extensive line graphics. Standards already exist in this area (such as GKS) and there is no reason for using different interfaces for line graphics on line oriented graphics devices and image displays. However, it is recognized that integrating a line graphics system with this image display interface is not a trivial task (this is discussed further in section **??**), and so the ability to add simple annotation using the image display interface alone has been provided.

It is not intended that these routines be used by "systems" or "environment" software to provide a user interface on single user workstation systems with high resolution bit mapped displays and sophisticated window management facilities. We do not feel that it is wise to build a generalized set of window management functions into this interface because there are already several such systems in existence and it is an area of intense activity in both the academic and commercial worlds. For example, the X Window System[8] from MIT is available in the public domain and is receiving very wide support and being ported to a number of new systems. Many new applications are being written to use the X Window interface. Apollo, Sun, and Masscomp also have proprietary window systems, and Sun has a network window management system (called NeWS[9]) that is based upon the PostScript[10] language (used by many laser printers) that may be widely adopted by other vendors. We feel that we should strive to operate within these systems rather than try to replace them. Although this might seem somewhat backwards looking since workstations are clearly going to grow in importance for astronomical data reduction, this interface must be implementable on the traditional type of image display device, which is still the type of hardware available to the majority of astronomers.

1.3 Interface Level

The primary purpose of the interface is to make it possible to write applications software that is independent of image display devices. One might assume, therefore, that code above the level of the interface must be device *ind*ependent, and code below the level of the

[8]X Window System is a trademark of the Massachusetts Institute of Technology.
[9]NeWs is a trademark of Sun Microsystems.
[10]PostScript is a trademark of Adobe Systems Incorporated.

interface must be device *de*pendent. The former statement is true with some qualifications, but the latter statement is false.

Code written above the interface must be device independent in the sense that it contains no explicit knowledge of any particular piece of hardware. However it is not unreasonable to expect an application to be targeted at a particular class of device and to exploit the facilities that that class of device offers. If appropriate use of inquiry functions is made in order to obtain precise details of the hardware configuration, then we can reasonably expect to be able to run the application on some different model of display which offers either similar or more extensive facilities. However, if the application uses a facility that does not exist on another display and which cannot be reasonably efficiently emulated, then we cannot expect to offer all functions of the application on the simpler display.

Code which appears below the interface *may* be totally device dependent, and for a sophisticated device which has all the facilities envisaged by the this interface in hardware, this will be so. However, most real devices will require at least some facilities to be emulated in software running on the host computer. In many cases the code can be made sufficiently general to be useful for several models of display, however, the specification of the interface to this device independent code is outside the scope of this document. We expect that there will be attempts to develop such layered implementations in the future. The development of a layered implementation of the IDI, where the device specific kernel is reduced to a minimum, will further reduce the effort required to implement the interface described here on new devices.

1.4 Existing Standards

Before a new standard is proposed it must be shown that no other existing standard, either formal or de facto, adequately addresses the problem. In the field of computer graphics the two standards that are relevant to image displays are the Graphical Kernel System (GKS–IS/7942) and the Computer Graphics Metafile (CGM–DIS/8632). Both of these standards were examined in detail and were both found to be inadequate for astronomical image display applications. The Virtual Device Interface standard will also probably be relevant, but it is currently at too early a stage to be considered.

Graphical Kernel System. As mentioned above, GKS has some facilities that are applicable to imaging applications, notably the cell array primitive, the color index table concept, and an escape function that can be used to extend GKS capabilities. However, the GKS model of a graphics device is unable to cope with the flexibility of configuration that many image display models offer. GKS views a graphics device as a display surface of fixed characteristics, and cannot be extended to accommodate devices with a variable number of display surfaces of different dimensions without fundamentally altering the whole structure of GKS.

It is possible to view a single piece of hardware as several GKS workstations, but for most devices the different display memories interact in too many ways for such an approach to produce behavior that is sufficiently close to the standard to be acceptable. It also becomes impossible to express the relationships between two workstations mapped onto the same screen (for example, the relative offset between an image and a graphics overlay) in any meaningful way.

Computer Graphics Metafile. The CGM standard has a more flexible model of a graphics device than GKS but is designed for the storage and transmission of pictures; it contains no interactive concepts nor does it support the concept of a sequence of operations applied to an existing picture. The CGM may well find use as a way of transporting image files among graphics and image devices, but cannot support the interactive display and manipulation of those images.

2 Conceptual Model

Before a device independent interface can be designed, an idealized device which is capable of being emulated by real hardware must be described. Our model for a generic image display device is described here by starting with the simplest possible system and then adding new features to build up to the most complicated system envisaged. As well as describing the model on which the interface definition is based, this section will serve to introduce and define some of the terminology to be used. Particular attention should be paid to these definitions as different hardware manufacturers have, in many cases, used the same terms to describe somewhat different concepts.

2.1 Model of User Interactions

The most frequent uses made of image processing hardware in astronomy are for image display and interaction; either for browse or "quick look" activities or for display of intermediate and final products during data reduction and analysis. In the limit, a suitable device may provide nothing more than a way of displaying four bits or so of low resolution grey scale or pseudo-colour data. Some imaging applications can be accommodated with little more than this, but we expect that the typical facility will have one or more image planes of at least eight bits, the ability to produce graphics and other annotations overlaid on the image, and some means of interaction.

The following model defines the image display features most commonly exercised during astronomical image processing. Functions are ranked in rough priority order. The simplest class of image display devices will typically only support the first set of functions. More sophisticated devices will support the second and third sets of functions. A high-end system will be required if performance considerations make it desirable to provide the last set of operations in hardware.

1. Load and display an image, load color table, annotate the image.
2. Manipulate the image:
   (a) pan, zoom, scroll
   (b) alter color tables interactively or under program control
3. Interact with the image:
   (a) control cursor with pointing device (track ball, joystick, mouse, bit pad), read back pixel values, annotate pixels
   (b) blink two or more images
   (c) produce split screen display of two or more images
   (d) make "movies" of an image cube
4. Add annotation and graphics:
   (a) Annotation – labels, axes, wedges, notes
   (b) Graphics – overlay, special cursor shapes, markers

5. Get hard copy output – mono or pseudo-colour screen dump, mono or colour printer interface, metacode file output

6. Use special functions – computationally intensive tasks such as histogram generation, rubber sheet distortion corrections, rotation, histogram equalization, smoothing, filtering, convolution, deconvolution, rendering, shading

Note that all of the category 6 functions can be done in software, but typically at some substantial computational cost. Many of the high-end display processors have video-rate hardware to perform some or all of the special functions, but these capabilities are often not used, largely for reasons of portability and cost. The presence of such facilities are not considered further within the interface, but an escape mechanism is provided that will allow use of such facilities if they are available.

2.2 Simple System

The simplest device that can be usefully employed as an image display consists of a pixel memory or frame store and a video look-up table that translates the data values stored in the pixel memory to gray-scale intensities or colors on a TV monitor.

On some devices, the data values in the pixel memory do not address the video look-up table directly, but instead address a look-up table known as an intensity transformation table (ITT). The contents of the ITT are used to select the video look-up table entries. ITTs are included in the simple model even though no such table exists in many real systems. A system without an ITT can be viewed as having an ITT with fixed contents (which is also the assumed default).

The *pixel memory* consists of a rectangular array of memory locations, each containing an integer in the range 0 to $z_{max} - 1$. The data range that can be represented in the memory can also be described in terms of the number of bits per pixel, referred to in this document as the *memory depth* D. Clearly

$$z_{max} = 2^D$$

The data value z stored in an element of the pixel memory is translated into a color on the screen as follows:

1. The data value present in an individual pixel memory is used as the index into the intensity transformation table. The intensity transformation table is a table of z_{max} entries numbered 0 to $z_{max} - 1$, with each entry containing a integer in the range 0 to I_{max} (I_{max} is the maximum intensity value on the device). The default ITT mapping is $I = z$.

2. The value in entry z of the ITT selects the index of an entry in the video look-up table (VLUT). The VLUT has I_{max} entries numbered 0 to $(I_{max} - 1)$, with each entry containing three values representing a red, green and blue intensity.

3. The R, G, and B values in entry I of the VLUT determines the color of the spot on the screen.

This is illustrated in Fig. 1.

The numbers stored in ITTs and VLUTs are integers, but in order to encourage device independence the interface routines pass them as real numbers in the range 0.0 to 1.0.

2.3 Multiple Memory Systems

The next level of sophistication is for the display to have several pixel memories (either provided in hardware or through some sort of configuration process). The memories need not be the same size, either in depth or resolution (number of pixels in x and y), and each memory is assumed to have its own intensity transformation tables. If a memory has more than one ITT associated with it then the one to be used when displaying the memory has to be selected. A default ITT/memory binding is specified for any given configuration.

Many systems also have multiple video look-up tables. Unlike ITTs these do not have a fixed association with memories and the number of VLUTs is not related to the number of memories. Before the contents of any memory can be displayed on the screen, it must be associated with a particular VLUT. We assume that the data path through the ITTs and VLUTs can be changed, and that either or both of these may be switched out of the display path. This is equivalent to loading the tables so that each input value maps to itself. Furthermore, we assume that the output of a specific image memory can be directed to one or more video output guns; often referred to as DACs (digital to analogue converters). It is by specifying whether a set of images are to be presented to all three DACs or bound one-for-one to individual red, green and blue DACs that the choice between pseudo-colour and true-colour displays can be managed. Again, each configuration supports a default binding between VLUTs and memories.

2.4 Overlays

Graphic and text overlays provide a convenient way of annotating displayed images without overwriting parts of the image data and without having to use video look-up table entries for line and text colors. They also allow the image colors to be manipulated without disturbing the annotation. Overlay memories are essentially the same as image memories except that pixels having the value 0 are transparent rather than opaque, and thus only the written portions of the overlay memory obscure the underlying image data.

Although overlay memories are not intrinsically different from image memories, they are often implemented quite separately in hardware and cannot be regarded as interchangeable with image memories, even when they are allocated from a common pool of memory. Overlay memories often have only one or a few bits per pixel. Their associated VLUTs also usually only have two possible intensities per color, either 0 or 1, and may be restricted to primary colors only. Many devices do not allow image data to be written to graphics overlays (nor would it be very useful to do so on one bit memories). Text overlays sometimes have peculiar properties: certain devices only allow text to be written outside the main picture area of the display, and others scroll the text automatically with no control over its placement.

For these reasons, overlay memories have to be handled rather differently from image memories. In particular, they are assumed to have VLUTs with defined initial contents which on some devices cannot be altered. For text overlays the result of any operation will be very hardware dependent; all that can be guaranteed is that text will be written somewhere on the screen.

2.5 Memory Visibility

When a device has several memories a means must be provided to specify where on the screen and at what time their contents should be displayed. Therefore, each memory has associated with it a position on the screen and a visibility flag. In general there are restrictions on how many memories of a particular type can be displayed simultaneously, and on how the memories may be positioned on the screen. It is often the case that several overlapping overlay memories can be displayed at once, but that image memories cannot overlap with one another. Some devices support the display of only one image memory at a time.

This wide functional variability is handled by allowing the visibility of a set of memories to be specified with a single function call; if the selected memories cannot be made visible concurrently by the hardware then an error status is returned. The visibility of memories not in the specified set becomes undefined and previously visible memories may be come invisible if the hardware demands it.

2.6 Pan, Zoom, and Scroll

Many devices have hardware facilities for enlarging (or possibly shrinking) the picture and moving it around the screen, often under the control of interactive devices such as track balls and joysticks. Unfortunately, different manufacturers use the same terms to describe facilities with small yet significant differences. It is important to distinguish between operations that affect the display of the whole picture and those which affect only the display of the contents of a particular memory.

The terms *pan* and *zoom* will be used to refer to the positioning and enlargement of the entire displayed picture, possibly derived from the contents of several image and overlay memories. One *pans* the display window in order to view different parts of the picture (including portions that may not fit on the image display screen). This operation is sometime described as 'roaming'. Zooming is allowed on each individual memory, but zooming by different factors in x and y is not supported by the interface.

The term *scroll* will be used to refer to the movement of one memory relative to another as displayed on the screen, for example when moving an overlay across an image. Whether or not the picture wraps around as it is scrolled depends on the capabilities of the hardware and on the level of the software implementation.

2.7 Interaction

Most image displays are equipped with devices that allow the user to interact with the picture or the applications program using the display. The devices commonly encountered are a pointing device such as a track ball, joystick, or mouse, that can be used to move a cursor around the screen (but has other uses as well), and some buttons or keys that can be used to trigger events or select among alternatives. Where these devices do not exist some sort of emulation is usually possible from a terminal keyboard.

We have adhered to the GKS model of a user interaction with its concepts of measures, triggers, and echos, but have not adopted the GKS interface for this model because it lacks the necessary flexibility. The GKS interface supports application programs which need to obtain measures. However, with image display devices the interaction is often used to govern the appearance of the image itself (*e.g.*, zoom, pan, contrast stretching). We have also allowed for a more flexible association of *interactors* (any of a class of interactive devices) with *objects* (the visual effect controlled by the interactor) and the *trigger* used to terminate the interaction.

For example, in order to turn on a cursor, allow the user to interactively reposition the cursor, and then read the cursor position, the following sequence of operations would be required:

1. Display the cursor (set cursor visibility *ON*).
2. Bind the cursor (the *object*) to the track ball (the *interactor*).
3. Specify which button device (the *trigger*) should terminate the interaction.
4. Initiate the interaction.
5. Wait for the user to terminate the interaction (*i.e.*, fire the trigger).
6. Read the cursor position.

More esoteric operations, such as "rotating" the color table with the track ball, can be handled in exactly the same way simply by using a different *object/interactor* binding. Also, more than one *interactor* may be active at one time, and more than one *object* can be bound to the same *interactor*.

2.8 Coordinates

In any graphics system the coordinate systems are of central importance. Since this interface is only concerned with driving hardware and not with any higher level picture modeling concepts, only two coordinate systems are needed. These are *memory coordinates* for reading and writing information to and from memories, and *screen coordinates* for specifying where on the screen a memory should be displayed.

Both of these coordinate systems have the origin in the bottom left hand corner; x increases to the right and y increases upwards. Memory coordinates are really pixel numbers rather than a true coordinate system and are intrinsically integer. Memory coordinate $(0, 0)$ refers to the bottom left pixel. It is not meaningful to ask if this refers to the center of the pixel or the bottom left corner of the pixel; a memory coordinate simply identifies a pixel memory location or memory cell address.

In contrast, screen coordinates are a true coordinate system; the units are the smallest size at which a pixel can be displayed, but it is meaningful to talk of fractions of a pixel in screen coordinates. Although one could imagine positioning a memory with a finer resolution than one unit, this is rarely permitted by the hardware, and these interfaces always pass screen coordinates as integers. Screen coordinate $(0, 0)$ is the bottom left corner of the screen, where the bottom left corner of pixel $(0, 0)$ would be displayed if the memory just fit on the entire screen. The position of a memory on the screen is given by the screen coordinate of the bottom left corner of pixel $(0, 0)$. Screen coordinates conceptually extend to infinity in both positive and negative directions.

The various coordinate systems with two memories displayed on the screen are illustrated in Fig. 2.

2.9 Mixed Image and Line Graphics

Image display devices are often used for line graphics, and it is most desirable to use the same graphics interface for image display devices as is used for conventional graphics devices. If it were acceptable for the device to be used either as a graphics device *or* an image device at any one time, there would be no particular problem. However, if the image display is to be used for both images *and* line graphics concurrently (more elaborate than the simple annotation facilities described in section **??**) some may difficulties arise.

The first problem is to ensure that the line graphics software and the image display software cooperate in both controlling the device and in sharing the available resources such as image memories. The ideal solution is for the line graphics device handler to use the image interfaces to drive the device. Internal conventions will need to be established to ensure that direct calls to the image interfaces by applications programs do not subvert the operation of the line graphics package.

The second problem is to ensure the precise registration of line and image graphics. The image display interface defines precisely which pixel will be written to by any operation and where this pixel will appear on the display screen, but this is by no means always true for line graphics systems (for example, the GKS standard does not specify precisely how world coordinates map to device coordinates). Therefore, the relationship between world coordinates and device coordinates will have to be precisely documented for any line graphics system that is to share a device with the image display interfaces.

Because it is not generally possible to modify typical GKS implementations, and because it is unlikely that GKS (or any other standard) will be extended to accommodate the needs of image display, we have chosen a simpler alternative. The image display interface specifications require that two simple routines be provided to accommodate line graphics and text on both image and overlay memories. These routines may be used to implement an interface for a local standard graphics package, or conversely, may be implemented using the standard graphics package. The possibility of extending the interface with a bit-map or Postscript graphics/text front-end is not precluded.

3 Interface Overview

This section describes the Image Display Interface (IDI) functions and their parameters in general terms, with the emphasis on how the functions relate to one another. Some remarks on the philosophy behind the interface design, and the reasons why some of the choices were made as they were, are also included.

When the name of an interface function is used it will be set in bold type (e.g.,**Open_Display**) and the names of function parameters will be set in slanting type (e.g., *display identifier*). The actual subroutine calling sequences and precise details of the parameters are presented in the next section.

3.1 Device Selection and Control

Before a particular display can be used it must be "opened" or "connected" by calling **Open_Display**. This function tells the device dependent software (i) what type of display is to be used, so that the correct set of commands can be used to control it, and (ii) the operating system device name that must be used to access it. This information is supplied by the *device name*. On a basic implementation built for a system with only one image display this could just be the type of display (*e.g.*, Gould or ARGS), but for an implementation that supports several displays, perhaps of different types, the name will probably be used as a key to a database containing details of all the displays on the system. Different names may also be used to select the same piece of hardware configured in different ways; this is discussed further in Section **??**.

Once a device has been opened, the applications software must have some way of referring to it subsequently. This is done with the *display identifier*, an integer returned by **Open_Display** and which appears as an input parameter to all other interface functions. If follows that **Open_Display** must be the first IDI routine called by an applications program.

Contrary to the GKS design, the IDI interfaces do not require that opening a device forces a reset of the device; a separate **Reset_Display** function has been provided for that purpose. The reason is that users may wish to move among applications packages, or between display and computational tasks, without having the open force a device reset and subsequent reloading of the displayed images. When the use of **Reset_Display** immediately after **Open_Display** is not appropriate an application must take the state of the display "on trust" and may fail to work correctly if the display has been corrupted by an aberrant application. It is assumed that most implementations will maintain a data structure that will be used to pass information about the state of the display among applications. If this is not the case, or if the information is incorrect, a call to **Reset_Display** must be made to restore the device to a known state.

Output to the display may be buffered to improve performance; **Update_Display** ensures that all buffered output has been sent to the display. An implicit update call is performed before every function that invokes interaction with, or input from, the display. This function is commonly referred to as "buffer flush"; "update" was chosen to conform with GKS terminology.

Close_Display updates the display, updates the run-time device state data structure, releases all resources used for controlling the display and relinquishes control of the device. Once a display has been closed it cannot be referred to again unless it is re-opened with a another call to **Open_Display**.

All functions except one have an integer *error indicator* argument (in some language bindings this may be a function return value rather than an explicit argument) which is set to zero if the function completes successfully, some value greater than zero if a severe or fatal error has occurred, and some value less than zero if the function was partially successful. An example of the last condition would be if the **Polyline** function drew the polyline but not in the requested *linestyle* (because the device did not support it). So that meaningful error messages can be constructed, the **Get_Error** function is provided to convert error numbers to text strings. This is the one function without an error return parameter; if called with a non-existent error numbers, an empty string is returned.

3.2 Device Configuration

On some of the more sophisticated displays the configuration (number of memories, number of graphics planes, resolution, *etc.*) can be modified under software control. Therefore, before the device can be used the specific configuration to be used must be agreed upon between the applications software and the device driver. Frequently the configuration rules for a specific device are too complicated or esoteric to be parameterized and passed to the application program; similarly the requirements of the application cannot easily be described to the driver software. We have chosen to define the interface such that configuration parameters can be queried and at least one, and possibly more than one, configuration is made available to the applications program.

To simplify the problem of defining device configurations at least one standard device configuration for each device must be defined and be available by default. A program may, through the use of several inquiry functions, determine the characteristics of the configuration. If other configurations have been defined the program may examine each of them in turn and select the one most appropriate for the task in hand. It is therefore up to the implementor of the IDI library to provide a suitable set of configurations. By defining several names for the same physical device, but with different default configurations, the choice of configuration can be placed in the hands of the user rather than the applications programmer if appropriate.

The **Query_Device_Characteristics** function returns the basic physical parameters of the device (such as the display resolution, the number of look-up tables, *etc.*) and the number of available configurations. The **Query_Defined_Configuration** and **Select_Configuration** functions are used to determine the characteristics of specific configurations and then to select the one which the application wishes to use. Query calls with -1 as the *configuration number* may be used to determine the characteristics of the currently selected configuration.

Some hardware devices require a hard reset to change the device configuration. Thus, it *cannot* be assumed that image memory data loaded when a configuration change is made will survive the change. A call to **Reset_Display** may be issued by any program that wishes to guarantee the device state before proceeding.

More detailed information about the characteristics of a device can be determined with **Query_Capabilities_Integer**. The integer argument *capability* specifies the characteristic being requested, and the result is returned as an array of integers. This function can be used to obtain, for example, the range of zoom factors available, or a list of the cursor types supported by the device. The routine **Query_Capabilities_Real** is also defined for some values, and a corresponding string function could be defined if required.

3.3 Memory Control

The display of memory contents on the screen is controlled by five functions: **Set_Memory-_Visibility**, **Write_Memory_Scroll**, **Write_Memory_Zoom**, **Select_Memory_Look-up_Tables**, and **Select_Display_Path**. Each memory in a configuration has an associated *memory identifier*. A list of valid *memory identifiers* can be obtained using **Query-_Defined_Configuration**. All memories (image, graphics, overlay) have unique identifiers. This identifier is used to select which memories functions operate upon.

Set_Memory_Visibility makes a specified list of memories visible; if the memories specified cannot be made visible simultaneously by the hardware it is considered an error. The visibility of memories not in the list is undefined; this means that memories that were visible and are not specified in the most recent call to **Set_Memory_Visibility** may or may not be visible, depending on the capabilities of the hardware. This allows applications programs to either demand that a particular set of memories be visible or to specify a priority order and accept the best that the hardware can achieve.

The position of the image on the screen is set by **Write_Memory_Scroll**, where *position* specifies the position in screen coordinates of the bottom left corner of the bottom left pixel; the position can lie outside the boundaries of the screen so that all or part of the memory is invisible. Memories can be zoomed with **Write_Memory_Zoom**; the *zoom factor* is a integer and applies to both x and y; positive values enlarge the picture and negative values shrink it. If a memory is made visible without **Write_Memory_Scroll** and **Write-_Memory_Zoom** having been called, it will be displayed at screen position (0,0) with a zoom factor of one. If the hardware does not allow the requested position and zoom to be realized, the closest possible approximation should be used; the meaning of "closest" is implementation dependent.

Both the zoom and scroll functions allow a list of memories to be set to the same position and zoom factor so that the effect of zooming and panning can be achieved with one call. If this were not so, then a device which was only capable of zooming and panning the whole display would be sent large numbers of redundant instructions.

The function **Read_Memory_Scroll_and_Zoom** is used for reading the current scroll and zoom settings. This function is needed for two reasons: (i) the application may wish to know the actual settings since they may differ from the values requested, and (ii) scroll and zoom may, at times, be under the direct control of the user via some interactive device.

The function **Select_Memory_Look-up_Tables** specifies which video look-up table (VLUT) and intensity transformation table (ITT) should be used when mapping a memory onto the screen. VLUTs are selected by *VLUT identifiers* and ITTs are selected by *ITT number* (and implicitly by the *memory identifier* since ITTs are not interchangeable between memories). It is an error to make a memory visible before it has been associated with a look-up table.

All configurations must contain a memory number 0 and a VLUT number 0, and every memory must have an ITT number 0.

A second function **Select_Display_Path** is defined to allow the path from memory to video gun as described in Section **??** to be placed under program control. It is assumed that the ITT and VLUT to be used have already been selected and loaded with values; **Select_Display_Path** is called simply to switch them into the hardware display path. This function may also be used to switch between pseudo-color and true color (RGB) modes, (where the hardware is capable of this) based upon the binding between the VLUTs and the actual color guns. Where it is necessary to perform a hard device reset in order to effect these changes this function may return a 'reset required' error code; the function would then have to be handled as a configuration option.

3.4 Image I/O

Image data is written into a pixel memory with **Write_Memory**. By default data is written from left to right starting at the position given by the *position* argument which is an offset from the bottom left hand corner of the current I/O transfer window (see below). When the right hand edge of the transfer window is reached, pixel writing continues on the next row up at the left hand edge of the window. The *array size* argument gives the total number of pixels to be written. The image data itself is an array of integers in the range 0 to z representing the pixel values to be written into the memory. Any values that exceed the maximum value that can be stored in the memory ($z_{max} - 1$) are truncated so that $\mathrm{mod}\,(z, z_{max} - 1)$ is actually stored.

Image and graphics memory data may be packed into integer words in a variety of different ways, depending on application characteristics and other choices such as efficiency of data transfer or storage. The read and write interface routines are capable of handling data whether it is packed one pixel per integer or many. *Packing factor* and *data depth* (in the memory) are separately specified so that the data can be handled as required.

Efficiency is a prime consideration when storing and manipulating image data on account of the quantity of data involved (maybe several megabytes for a single image). The most efficient data type or structure to use in any particular case depends on the range of data values stored in the pixel memory. It may be necessary for applications programs to have several versions of the image i/o functions, each supporting a different representation of the image data.

A *transfer window* is a rectangular subset of the image display memory. Successive pixel values from the image are loaded into successive memory locations in the transfer window. Many devices support a transfer window to improve I/O speed. Other devices require this concept because the I/O window is actually smaller than the image memory dimensions. The use of transfer windows allows images to be transferred to the display a piece at a time so that very large images can be handled without requiring excessive amounts of host memory. The **Set_Transfer_Window** function allows the application to set the window size and position for I/O transfers. By default the transfer window is set to the full memory size returned by **Open_Display** or whenever a new configuration is selected with **Select_Configuration**. For those devices where the largest possible transfer window is smaller than the memory it is set accordingly and the maximum size may be determined with the **Query_Defined_Configuration** function. **Set_Transfer_Window** can also be used to have the image loaded from top to bottom rather than from bottom to top for cases where the data is organized in that fashion.

The contents of an area of a pixel memory can be read with **Read_Memory**. **Read_Memory** is the complement of **Write_Memory** with the additional ability to transform the data via the intensity transformation table. Whether or not this transformation is done is controlled by *ITT flag*. Support of this feature is optional; for devices with fixed ITTs the state of the flag is clearly irrelevant.

Clear_Memory is used to erase the contents of a memory. Where the hardware supports it, the memory may be filled with a value other than zero as specified by *background value*.

If the hardware does not support this function the background value may be ignored. The same function is used for image, graphics, and text memories; on graphics and text memories it is assumed that zeroing makes them transparent.

3.5 Graphics

Only limited functions are provided to create graphics and annotation. It is assumed that these functions are used for overlay plots or for simple annotation of images. If more complex operations are desired it is suggested that the GKS or some other graphics interface should be employed. The simple functions included in the interface may be supported by hardware functions or by simple software implementations. It is assumed that graphics and text memories that are used for annotation are transparent except where they are written on (*i.e.*, non-zero), but that they may be otherwise treated exactly like any image memory.

The **Polyline** function connects a series of points with straight lines. Coordinates are specified in *memory* coordinates. Line color can be specified and optional control over line style is provided.

Text strings may be plotted with **Plot_Text**. It is intended that hardware characters should be used if available, and even if control of the height, text path, and angle are available, support for these parameters is optional. The closest approximation that the hardware can manage should be used. If no hardware characters are available, then software generated characters must be used, but a simple raster font of height 1, angle 0, and text path RIGHT is sufficient.

The meaning of the *color* parameter depends on the type of memory being written to; for image memories it is just the pixel value used when drawing the line or text and the actual color depends on the contents of the associated look-up tables. However for graphics memories where the contents of the VLUTs are often fixed, the VLUTs are assumed to have defined initial contents so that programs can use *color* in a predictable way. On devices that have fixed VLUTs the interface library must handle the mapping of the color number to the actual pixel value written into the memory. If the VLUTs can be changed programs may modify the mapping, but care must be taken to preserve portability.

For text written to text memories, the *color* may be interpreted as a video attribute if color is not available.

3.6 Look-up Table I/O

Video look-up tables and intensity transformation tables are read and written with two pairs of symmetrical functions: **Write_Intensity_Transformation_Table**, **Read_Intensity-_Transformation_Table**, **Write_Video_Look-up_Table**, and **Read_Video_Look-up-_Table**. The reading and writing routines have identical parameter lists, except that the table parameter is an input parameter in one case and an output parameter in the other. In the VLUT I/O routines each table entry consists of three real numbers in the range 0.0 – 1.0, whereas in the ITT routines each table entry consists of a single real number. VLUTs are identified by *VLUT*, and ITTs are identified by *memory identifier* and *ITT number*. The functions allow a contiguous block of entries, which can be as large as the whole table or as small as a single entry, to be read or written as specified by *start position* and *number of entries*.

The video look-up tables for overlay memories are not fundamentally different from those of image memories. However, the number of entries is typically small, often only two, and the range of possible values that an entry can take is also small, again usually only two (0.0 or 1.0). The same functions used for regular VLUT and ITT color values are used to write and read overlay color tables. However the graphics functions assume that only a limited color range is available and that these colors are established by default in the VLUTs for the overlay memories. If the VLUTs are altered then the *color* argument will not select the defined color, but whatever color has replaced it in the VLUT.

3.7 Zoom and Pan

Zooming the entire display and panning the visible window around the picture are treated as different operations from zooming and scrolling individual memories. This is most usefully done under the control of the user with some interactive device, but two functions for setting and reading back the zoom and pan have been defined (**Write_Display_Zoom_and_Pan** and **Read_Display_Zoom_and_Pan**) so that direct control by applications programs is possible.

3.8 Cursor Control and Region of Interest

Control is provided over one or more cursors that may be visible on the screen at any one time. These cursors may be initialized to either a fixed shape or a programmable shape by use of the **Initialize_Cursor** function. The cursors will be visible only when a call to **Set_Cursor_Visibility** has been made. Cursor positions may be read or updated via calls to **Read_Cursor_Position** and **Write_Cursor_Position**. Interaction with the cursor is not handled by these routines, but by a class of interactive functions that are described in the next section.

A common use of cursors is to indicate a rectangular region of the screen or *region of interest* (ROI). Many devices provide a better way of indicating such an area than just marking opposite corners with a cursor, including ways for the user to modify the position and size of the area in one operation. So that these hardware facilities can be exploited, a set of routines analogous to the cursor routines have been defined for the creation and manipulation of a rectangular ROI. The way in which the ROI is indicated is not defined, it may be done for example, with only two simple corner markers, with dual cross-hair cursors, or with a rubber-banded box around the region. Any of these are suitable as long as the region is clearly indicated.

3.9 User Interaction

Before the user can interact with the display some interactive device (an *interactor*), such as a track ball or button box, must be bound to some aspect of the display (an *object*), such as a cursor or a memory, and to the form that the interaction should take, such as move or zoom. Some event (a *trigger*) that will terminate the interaction must also be specified. This is done with the **Enable_Interaction** routine; on some devices which can handle interactions independently of the host computer, this will also start the interaction process. On devices which need to involve the host in order to perform the interaction, the interaction is actually started by calling **Execute_Interaction_and_Wait**. In the former case this routine simply waits for the trigger event to terminate the interaction.

Interactors are selected by their type (*e.g.*, *evaluators*, which return a single number, or *locators*, which return an x, y displacement) and by a unit number which selects between devices of the same type. *Objects* are similarly identified with the unit number representing a memory identifier, VLUT identifier, cursor number, *etc.*, as appropriate. The argument *interactive operation* selects the type of operation that is performed on the object. A special case is when *interactive operation* is set to 0, which allows some application-specific code to be executed each time the interactor is read. When this operation type is selected, **Execute_Interaction_and_Wait** returns if an interactor (locator or evaluator) has been enabled or if an enabled trigger has been fired, even if the exit trigger has not been fired. The applications program can read the interactor (with **Get_Locator_Displacement** or one of the **Get_Evaluator** functions), perform some applications specific operation on the display (or even on some entirely different device) and then call **Execute_Interaction_and_Wait** again, continuing to loop in this fashion until a non-zero exit trigger status is returned. This mode of operation is likely to give uneven response times on multi-user systems and to impose a considerable load on the host. This load may be reduced if the hardware is capable of raising an interrupt when an interactor changes state rather than having to be polled by the host. The behavior of **Execute_Interaction_and_Wait** for *operation type* = 0 is summarized in the table below. The first two columns indicate the locator or evaluator status and trigger status *on entry into the routine.* The exit trigger status refers to the trigger status *within the routine.*

locator or evaluator	trigger	exit trigger status	exit from routine?
enabled	any	any	yes
not enabled	enabled and fired	any	yes
not enabled	enabled, not fired	fired	yes
not enabled	enabled, not fired	not fired	no

Behavior of Execute_Interaction_and_Wait
for *operation type* = 0

Interactor type specifies the type of interactive device (locator, real evaluator, integer evaluator, logical evaluator or switch, character evaluator, and trigger). Other *interactor types* may also be defined but would be implementation dependent.

Stop_Interaction terminates an interactive session and cancels the binding of interactors to objects. It allows an interaction that is taking place asynchronously to be terminated cleanly in response to an outside event (such as some error unrelated to the interaction which causes the application to exit).

Query_Interactor_Description returns a text string that contains a description of an interactive device. It may be used to supply instructions to the user without embedding device dependent information in applications programs.

3.10 Miscellaneous Functions

There are a number of somewhat specialized functions which are sufficiently general to be defined in a device independent interface. These optional functions often exercise features of particular hardware devices and may be difficult to implement on other devices. They are included so that applications can use them in a way that can be ported to other devices with the same functions, but they do not need to be implemented except where required by applications programs.

Blink_Memories covers two common applications (i) blinking two images to reveal small differences between them, and (ii) displaying a sequence of images to produce a "movie". Each memory in the *list of memory identifiers* is displayed for the *blink period* in turn until the operator takes some display (or implementation) dependent action to terminate the sequence.

Set_Split_Screen allows two or more images to be displayed on the screen at the same time. Three different split modes are provided: two memories split vertically or horizontally, and four memories, one in each quadrant. Some devices cannot support all of these modes, some can support others as well, but only these three common configurations are supported. Much more complex screen arrangements are possible, but these are left to the realm of workstation window management systems, as described in section **??**.

Create_Snapshot is used to obtain a snapshot of the picture that appears on the screen. It is intended to provide a "what you see is what you get" style of hardcopy capability, *not* a copy of all of the data that has been loaded into the display. In general the information necessary to accurately reconstruct the displayed picture cannot be obtained by the applications program because too many device dependencies are involved. A representation of the picture is returned as either a pixel array representing a monochrome image, or three arrays representing the R, G, and B components of the picture. This information can then be processed into a form appropriate for the available hard copy devices.

Set_Intensity_Bar_Visibility displays an intensity bar or ramp on the screen; the position and detailed appearance is entirely hardware dependent. When emulated in software, provision should be made for restoring any data that was overwritten when drawing the bar.

Many of the parameters that define a given implementation and configuration are available from the various Inquiry routines that are provided, and the parameter escape functions give access to additional information. The **Diagnostic_Routine** is provided to permit a diagnostic dump function that will display implementation-specific information on the selected display device. It will typically be used for debugging purposes and is only required to provide an ASCII dump of relevant information to the screen. Although it could be argued that such a routine does not belong in a applications oriented interface, it has been included so that porting code between IDI applications can be made a little easier.

3.11 Workstation Interface

For some devices, and most particularly workstations, it is desirable to support programming modes that provide a much higher level of flexibility than is available in the interface routines described so far. An additional set of routines is provided that can be used to allow dynamic definition of new memories (effectively creating new device configurations), thus exploiting some of the capabilities of modern workstations.

There are really two modes of operation that are viable for workstations within the IDI model: (i) use whatever window manager is available on the workstation to create a new window on the screen and treat that as a separate virtual display; or (ii) use the routines described below to create new memories in the current virtual display and manage them within the single IDI display. Either mode (or both) may be utilized in an IDI implementation on a workstation.

The function **Enable_Configuration** is used by an applications program to initialize the negotiation for memory resources. A simple routine called **Allocate_Memory** is provided to allow memories to be dynamically created. When the negotiation for memories is completed, the **Stop_Configuration** routine is used to fix the new memory configuration and associate it with a new *configuration number*. In order for an application to utilize this new memory configuration, **Select_Configuration** must be called using the new *configuration number*.

The function **Release_Configuration** is used to release the memory allocated by in this dynamic configuration process. After a configuration is released, the *memory identifiers* assigned by **Allocate_Memory** and the *configuration number* assigned by **Stop_Configuration** become invalid.

Note that these routines may be implemented (with some significant amount of work) on devices that do not support the standard workstation windows. If they are implemented using an available workstation window system, peculiar side-effects may occur if these windows are manipulated by the window manager, but this is unavoidable given the simple approach adopted. We strongly suggest that the IDI's be used to control image display within an existing window manager, and recommend against attempting an independent window management implementation within the IDI functions.

3.12 Escape Functions

It is inevitable that many models of display will possess hardware features that are not utilized by this interface, and although the use of such features will make applications code non-portable, some way of accessing them must be available if these interfaces are to be accepted. Therefore two escape functions (**Get_Escape_Parameter** and **Put_Escape_Parameter** for reading and writing device and implementation dependent parameters have been defined. Use of these functions is strongly discouraged except where absolutely necessary.

4 Interface Specifications

The set of interfaces defined in the body of this document is rather large, but is felt to define a complete functional set of interfaces for use across a broad range of image devices. A reduced set of these interface routines is sufficient to display an image, but the complete set as defined here is required to provide the capabilities commonly required by existing astronomical applications. The routines that are defined must all be provided for an implementation to be considered complete, although some may only be stub routines.

It is acceptable to provide implementations where only a single default configuration is supplied, but it must be possible to use the query and select functions to establish this configuration and its parameter values.

Although most of the functions only control one aspect of the display, they do interact to some extent in that interface and memory control functions must be called before any of the display or I/O functions can work. Also note that there are interactions between the zoom and pan functions and the cursor and window functions. Implementation details and limitations in the hardware may alter the exact semantic actions that can be achieved, but the intended actions, as described here, should be used as the ideal. Any instance of the interface that varies from the standard (because of hardware or implementation limitations) must be clearly documented.

The interfaces are grouped into three sets:

1. The *basic* set of routines, which includes the routines absolutely necessary for a minimum IDI implementation.
2. The *extended* set of IDI routines, which consists of the routines which may have to be emulated on lower level image displays.
3. The *optional* set of IDI routines, which may be easily implemented on the appropriate hardware but which may be difficult or impossible to emulate in software.

A complete IDI implementation consists of both the basic and the extended set of the IDI routines.

The image display facilities seen by an applications program will not only depend on the facilities offered by the hardware being used, but also on the ingenuity of the writer of the interface software. It would probably be possible to provide all the the functions described here on the simplest hardware, but the resulting performance penalties are likely to be such that the system would be almost useless.

Exactly what is, and what is not, emulated will depend on the individual applications. For example, if the aim is to support new applications written for some particular model of display, then exploiting the available hardware is the primary goal and very little software emulation would be required. However, if the aim is to transport applications written for more sophisticated devices to a cheaper system, then a considerable amount of emulation may be necessary. In particular, those routines labelled "optional" only need to be implemented when a definite requirement has been identified.

A minimal implementation would support just one default configuration consisting of one image memory with one fixed ITT and one VLUT. Then many routines (*e.g.*, **Select_Configuration**, **Select_Memory_Look-up_Tables**, *etc.*) become trivial to implement; they either do nothing or return an error. Also, many of the arguments of several functions can be ignored (such as *linestyle* in the **Polyline** function). Where programming effort is very restricted, those routines which are labelled "extended" can be mere stub routines that just return a 'not-implemented' error.

In the following function definitions the GKS conventions for calling sequences, subroutine and option names, and function were adhered to wherever appropriate. This has been done to maximize consistency between these two sets of routines. In some cases the resulting interface names or coding styles are not as pleasing as they might be otherwise, but these aesthetic considerations should not present any functional limitations. In some specific instances the GKS techniques have been consciously avoided because they were in conflict with either the desired function of the interface or with current practice in the astronomy field.

The routine specifications are presented in a language independent fashion but with terminology that will be familiar to the FORTRAN programmer. Argument data types are specified as *integer* (fixed point, FORTRAN type `INTEGER`), *real* (floating point, FORTRAN type `REAL`), *string* (FORTRAN 77 `CHARACTER` string), and *logical* (an integer where 0 represents 'false' and non-0 represents 'true'). Logicals have been mapped onto integers because some other programming languages do not have an explicit logical data type and there is a wide variation in the way different FORTRAN compilers represent logicals. The six character subroutine names have been derived from a two character prefix (`II`), a single character indicating the aspect of the display being operated on (*e.g.*, `M` for memory, `L` for look-up table) and a three character mnemonic for the type of operation (*e.g.*, `UPD` for update, `PLY` for polyline). The resulting names are undoubtedly somewhat cryptic but this is unavoidable if name clashes with other packages are to be avoided within the six character limit imposed by the FORTRAN 77 standard.

In C the data types should be interpreted as follows:

Integer	`int` for input arguments; pointer to `int` for output arguments
Integer()	pointer to array of `int`
Real	pointer to `real`
Real()	pointer to array of `real`
String	pointer to array of `char` (`NUL`-terminated)
Logical	`int` ($0 \Rightarrow$ false, non-0 $\Rightarrow$ true)

The *error indicator* is returned as the function return value (type `int`), not as an explicit argument.

For each routine we define a long name (set in large, bold-face type), a FORTRAN 77 name (in typewriter font), the category of the subroutine (basic, extended, optional), and the argument list. Each argument is specified as input (In) or output (Out), followed by its data type and a descriptive phrase. Arguments are given in the order in which they must appear in the calling sequence. Although no FORTRAN dummy variable names are specified for the subroutine arguments, we urge implementors to follow the FORTRAN 77 standard in using names of 6 characters or less.

Routines that require specification of one of a number of options, or which return a selection of one of a number of options, code such options as integers with defined symbolic names. This has been done to minimize the amount of character string handling in the interface. We suggest that implementors define these integer values with more useful mnemonics (using, for example, `include` files in C implementations, or even `INCLUDE` files in Fortran implementations if your host Fortran compiler supports them).

4.1 Device Selection and Control

Open_Display IIDOPN
basic

In	String	Device name
Out	Integer	Display identifier
Out	Integer	Error indicator

Open_Display allocates a display (or part of a display if one display is being treated a more than one logical IDI device) and initializes the state of the IDI software to mirror the state of the display. This requires that the state of the display saved by the preceeding call to **Close_Display** must be accessed. If this is impossible then the display will be reset to the appropriate default state and a warning error indication returned.

Device name identifies the physical device, the display type (Gould, ARGS, *etc.*) and the default configuration for that device. The IDI software is expected to know what devices are present on a system and what their physical device names are. These names will be implementation specific, but a suggested format for them is given in Appendix B.

Display identifier is used to refer to the device in subsequent calls to interface functions. It may be any value and should not be manipulated by applications programs in any way other than by passing it to another IDI function.

It is recommended that both the physical device configuration and the specific logical configurations supported by the implementation should be kept in a set of tables external to the actual support code. Interface implementations for a single device may choose to ignore this refinement, but more complete implementations should not.

Close_Display IIDCLO
basic

In	Integer	Display identifier
Out	Integer	Error indicator

Close_Display updates the display, releases all resources used for controlling the display and relinquishes control of the device. Once a display has been closed it cannot be referred to again unless it is re-opened with a call to **Open_Display**.

Close_Display must preserve the run-time state of the device for a later use by **Open_Display**

Reset_Display IIDRST
basic

In	Integer	Display identifier
Out	Integer	Error indicator

Reset_Display resets the display hardware and can be used to force the display to be in a "known" initial state. Device configuration changes may also need to exercise this function for those devices which require a "hard reset" to alter operating modes.

A call to **Reset_Display** should clear all of the memories, load the VLUT and ITT with linear tables, and leave the device in the configuration last selected (either by a call to **Select_Configuration** or by **Open_Display**.

Update_Display IIDUPD
basic

In	Integer	Display identifier
Out	Integer	Error indicator

Output to the display may be buffered to improve performance. Calling **Update_Display** ensures that all buffered output has been sent to the display. This function is called implicitly by any function that reads data from the display and by **Enable_Interaction**.

Get_Error IIDERR
basic

In	Integer	Error number
Out	String	Error message text
Out	Integer	Length of message text

Get_Error converts an integer error message number returned by an IDI function to character string that contains the text of the error message. If the *error number* is not defined, an empty string is returned.

4.2 Device Configuration

Query_Device_Characteristics IIDQDV
basic

In	Integer	Display identifier
Out	Integer	Number of available configurations
Out	Integer	Display size in x (pixels)
Out	Integer	Display size in y (pixels)
Out	Integer	Display depth (number of bits in DACs)
Out	Integer	Number of VLUTs in device
Out	Integer	Number of ITTs per image memory
Out	Integer	Number of cursors
Out	Integer	Error indicator

Query_Device_Characteristics is used to inquire about parameters that are not configuration dependent and that describe the basic characteristics of the physical device that is being used.

Display size refers to the largest area of memory that can be displayed on the screen, not the size of the pixel memories in the device. *Display depth* is the number of different intensity levels that the device is capable of displaying, expressed as the number of bits in the digital to analog converters.

Query_Capabilities_Integer

`IIDQCI`
basic

In	Integer	Display identifier
In	Integer	Capability
In	Integer	Size of output array
Out	Integer()	Output array
Out	Integer	Number of values returned
Out	Integer	Error indicator

Query_Capabilities_Integer is used to inquire about the device capabilities that are not configuration dependent and that describe the physical device (or image display station) that is being used.

The allowed values of *capability*, which is a symbolic integer, are listed in Appendix **??**.

Query_Capabilities_Real

`IIDQCR`
basic

In	Integer	Display identifier
In	Integer	Capability
In	Integer	Size of output array
Out	Real()	Output array
Out	Integer	Number of values returned
Out	Integer	Error indicator

Query_Capabilities_Real is used to inquire about the device capabilities that are not configuration dependent and that describe the physical device (or image display station) that is being used.

The allowed values of *capability*, which is a symbolic integer, are listed in Appendix **??**.

Query_Defined_Configuration

IIDQDC
basic

In	Integer	Display identifier
In	Integer	Configuration number
In	Integer	Memory type
In	Integer	Max number of memories
Out	Integer	Configuration mode
Out	Integer()	List of memory identifiers
Out	Integer()	Memory sizes in x
Out	Integer()	Memory sizes in y
Out	Integer()	Memory depths
Out	Integer()	Memory ITT depths
Out	Integer	Number of memories
Out	Integer	Error indicator

The possible configurations available on a device are numbered 0 through n. For each configuration, the *configuration mode* and a list of each of the three types of memories (image, text and graphics) can be obtained. A separate call is required for each type of memory with the *memory type* parameter set to the appropriate value. The allowed values for *memory type* are:

integer code	description
1	image memory
2	text (alphanumeric) memory
4	graphics memory

Allowed values of *memory type*

The parameter *memory type* is effectively a bit mask. Memories which must serve a dual purpose are indicated by summing the codes representing each function (so that, for example, a single memory which is to be used for both image and graphics would have an integer code of 5).

The subroutine returns a list of memory identifiers and, for each memory, its size, depth, depth of its associated ITTs, and the total number of memories of the specified type. If the number of available memories is greater than the value of the parameter *maximum number of memories*, only *maximum number of memories* memory identifiers will be returned. Note that in any one configuration no two memories have the same identifier even if they are of different types.

Configuration mode indicates the mode in which the device is set up [monochrome, pseudocolor, or true color (RGB)]. This mode reflects the internal state of the binding of the various VLUTs to video output paths. It is possible to use the **Select_Display_Path** function to change the VLUT bindings (and thus the display mode) after a configuration has been selected if the hardware is capable of changing color modes without a total reset or reconfiguration. The returned values are:

integer code	description
0	monochrome mode
1	pseudo-color mode
2	true-color (RGB) mode

Allowed values of *configuration mode*

A call to **Query_Defined_Configuration** with *configuration number* set equal to -1 will return information about the currently selected configuration (as selected by **Open-_Display** or by **Select_Configuration**).

Select_Configuration `IIDSEL` *basic*

In	Integer	Display identifier
In	Integer	Configuration number
Out	Integer	Error indicator

The predefined configuration identified by *configuration number* is selected. This function also sets up a default transfer window equal to the smaller of the maximum memory size or the maximum transfer window size.

4.3 Memory Control

Set_Memory_Visibility `IIMSMV` *basic*

In	Integer	Display identifier
In	Integer()	List of Memory identifiers
In	Integer	Number of memory identifiers
In	Logical	Visibility
Out	Integer	Error indicator

Setting a memory visibility to 'true' causes it to be displayed using the currently selected ITT and VLUT. Making a memory visible may have the effect of making other memories invisible depending on the capabilities of the device. It is an error to make a memory visible before it has been associated with a ITT and a VLUT. If the specified visibilities cannot be realized, an error should be returned; the visibility of memories not in the list is implementation dependent.

Write_Memory_Scroll IIZWSC
basic

In	Integer	Display identifier
In	Integer()	List of memory identifiers
In	Integer	Number of memory identifiers
In	Integer	x offset
In	Integer	y offset
Out	Integer	Error indicator

By default memories are displayed with pixel (0,0) at the bottom left corner of the screen. **Write_Memory_Scroll** allows one or more memories to be offset by an integral number of pixels. The offset is relative to the default position in screen coordinates: positive x and y values shift the memory upwards and to the right. The memory should be set to the nearest position that the hardware can achieve.

Write_Memory_Zoom IIZWZM
basic

In	Integer	Display identifier
In	Integer()	List of memory identifiers
In	Integer	Number of memory identifiers
In	Integer	Zoom factor
Out	Integer	Error indicator

By default memories are displayed with a one-to-one mapping between memory coordinates and screen coordinates. **Write_Memory_Zoom** allows the memory to be scaled by integer factors. Positive values of *zoom factor* cause memory pixels to be replicated on the screen; negative values cause only every n'th pixel to be displayed. Zooming leaves the position on the screen of the center of the memory unchanged. However, if one or more cursors are visible, the position of the cursor having the lowest numbered cursor identifier should remain unchanged. If a region of interest is visible, the center of the region of interest should remain unchanged. Cursors have priority over regions of interest. The memory zoom and position should be set to the nearest values that the hardware can achieve.

Read_Memory_Scroll_and_Zoom IIZRSZ
extended

In	Integer	Display identifier
In	Integer	Memory identifier
Out	Integer	x offset
Out	Integer	y offset
Out	Integer	Zoom factor
Out	Integer	Error indicator

Read_Memory_Scroll_and_Zoom allows the current position and zoom factor for a memory to be read. The values returned will be the actual values realized by the hardware which may be different from the values selected by the applications program using **Write-_Memory_Scroll** and **Write_Memory_Zoom**.

Select_Memory_Look-up_Tables

IIMSLT
extended

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	VLUT identifier
In	Integer	ITT number
Out	Integer	Error indicator

Select_Memory_Look-up_Tables selects the intensity transformation table and video look-up table to be used when displaying the contents of the memory. If *VLUT identifier* or *ITT number* is set to -1, the current VLUT/ITT selection is retained.

Select_Display_Path

IIDSDP
extended

In	Integer	Display identifier
In	Integer()	List of memory identifiers
In	Integer	Number of memory identifiers
In	Integer()	List of VLUT flags
In	Integer()	List of ITT flags
Out	Integer	Error indicator

Select_Display_Path selects the display path of the visible memory through the VLUTs and ITTs. Each video output path has three sections numbered 1 (R), 2 (G), and 4 (B). The values in *list of VLUT flags* specify the path to which each memory is to be bound. For pseudo-color displays each of the image memories is bound to all three DACs, so a flag value of 7 ($R+G+B$) is used for all memories. For true-color displays each of three memories is separately bound to a DAC, thus the flags will be 1, 2, and 4 for each of the memories in a set which are to contain, respectively, the R, G, and B components of an image. A negative value in the *list of VLUT flags* retains the current VLUT setting for that memory.

ITT flag specifies whether the ITT selected for each memory should be bypassed (as if it were loaded with a linear table) or not. The flag can take the following values:

integer code	description
> 0	Use ITT
0	Bypass ITT
< 0	Retain current ITT setting

Allowed values of *ITT flag*

For many models of display (*i.e.*, those with no true-color mode and no ITTs) this function will just return the error 'function not implemented'. If the display is capable of changing color modes, but only with a hard reset, the setting of the mode with this function should not be supported and an error of 'reset required' should be returned. Instead, the routine **Select_Configuration** should be used to select the mode.

4.4 Image I/O

Write_Memory `IIMWMY`
basic

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer()	Image data
In	Integer	Number of pixels
In	Integer	Data depth (bits/pixel)
In	Integer	Packing factor
In	Integer	Start position in x
In	Integer	Start position in y
Out	Integer	Error indicator

Write_Memory is used to load image data into a memory. Data transfers may be of any size: a pixel at a time, a line at a time, or a full image in one call. Data is assumed to be stored in memory as a single contiguous array and will be loaded into memory in the order received. This function can be used to write to any type of memory provided that the hardware supports the writing of image data into the specified memory.

The window boundaries that control how the data wraps from one pixel row to the next are controlled by **Set_Transfer_Window**; default settings are automatically defined with calls to **Open_Device** or **Select_Configuration**. If separate writes are used to load parts of an image the x and y data positions must be updated to indicate where each write is to start.

Packing factor describes the number of data items packed into a single integer word. For example, for a system with 32 bit integers there could be one 32 bit integer, two 16 bit integers or four 8 bit integers packed into each word. *Data depth* refers to the number of bits of data per pixel. If *data depth* is not equal to the depth of the memory being written to, each pixel value will either be truncated at the most significant end or extended with zeros to fit the memory depth. For example, if 8 bit pixels are stored one per integer word, *packing factor* would be 1 and *data depth* would be 8. If two 16 bit pixels are packed into a 32 bit integer word, *packing factor* would be 2 and *data depth* would be 16.

The use of *data depth* which does not match the memory depth may result in a considerable overhead for some models of display. In general only a limited number of data depths (*e.g.*, 1, 2, 4, 8, 16, 24, 32) will be supported.

Clear_Memory

IIMCMY
basic

In	Integer	Display identifier
In	Integer()	List of memory identifiers
In	Integer	Number of memory identifiers
In	Integer	Background value
Out	Integer	Error indicator

This function is used to erase the entire contents of a memory.

If supported by the hardware the memory may be filled with a value other than zero specified by *background value*. If the hardware does not support this function the background value may be ignored.

Read_Memory

IIMRMY
basic

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	Number of pixels
In	Integer	Start position in x
In	Integer	Start position in y
In	Integer	Data depth (bits/pixel)
In	Integer	Packing factor
In	Logical	ITT flag
Out	Integer()	Image data
Out	Integer	Error indicator

Read_memory is the reverse of **Write_memory** and the arguments all have the same meaning except *ITT flag*. When *ITT flag* is set to 'true' the values read are to be transformed by the intensity transformation selected for the memory. When *ITT flag* is 'false' it indicates that values are not to be transformed. Note that this function, like **Write_Memory**, can be used to read either image, graphics, or text memories where the hardware allows it.

Set_Transfer_Window `IIMSTW` *basic*

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	Load direction
In	Integer	x size
In	Integer	y size
In	Integer	Data depth
In	Integer	x offset
In	Integer	y offset
Out	Integer	Error Indicator

This routine is provided primarily for efficiency reasons so that display hardware can be used to optimize transfers. The default transfer window is established in the **Open_Device** call and is reset automatically by a call to **Select_Configuration**. The transfer window need only be redefined if the I/O window within an image plane is to be changed. Note that a given device may have large image memories but may only permit I/O transfers to a small section of the memory at a time. **Set_Transfer_Window** must be used to move the I/O window around in the writable image memory.

The *load direction* argument specifies whether images should loaded bottom up or top down:

integer code	description
0	load from bottom of screen to top
1	load from top of screen to bottom

Allowed values of *load direction*

4.5 Graphics

Polyline `IIGPLY` *extended*

In	Integer	Display Identifier
In	Integer	Memory Identifier
In	Integer()	List of x positions
In	Integer()	List of y positions
In	Integer	Number of (x, y) positions
In	Integer	Color
In	Integer	Line style
Out	Integer	Error indicator

Polyline connects the specified points with straight lines. The value of *color* determines the color of the line (in conjunction with the associated ITT and VLUT) in a way that depends on the type of memory being written to. For image memories *color* is the pixel value written into the memory to draw the lines. For graphics memories the value of *color* selects the color according to the following table:

integer code	description
0	background
1	black
2	white
3	red
4	green
5	blue
6	yellow
7	magenta
8	cyan

Allowed values of *color*

These colors will be displayed unless the VLUT associated with the memory has been altered by the applications program. The effect of setting the value of *color* > 8 is implementation dependent. If the specified color is not available, white should be used.

Line style is an integer in the range 1 to 4. The linestyles are the same as those defined by GKS:

integer code	description
1	solid line
2	dashed line
3	dotted line
4	dash-dotted line

Allowed values of *line style*

If the specified *line style* is not available, a solid line should be used.

Plot_Text IIGTXT

extended

In	Integer	Display identifier
In	Integer	Memory identifier
In	String	Text
In	Integer	x position
In	Integer	y position
In	Integer	Text path
In	Integer	Text orientation
In	Integer	Pixel color value
In	Integer	Text size
Out	Integer	Error indicator

Plot_Text is intended for plotting hardware text but devices without hardware text must provide a software font (probably a raster font). The text should be displayed with the nearest height available.

The *x* and *y positions* refer to the bottom left corner of the first character.

Legal values of *text path* are:

integer code	description
0	text runs from left to right
1	text runs from bottom to top
2	text runs from right to left
3	text runs from top to bottom

Allowed values of *text path*

The support of *text paths* other than from left to right is optional.

Text orientation is specified in degrees of rotation in a clockwise direction; support of orientations other than 0 is optional.

The value of *color* is interpreted as the text mode if the memory identifier refers to a text memory; otherwise it is the same as is defined for **Polyline**. When interpreted as a text mode, the following values are defined but the displayed values may be implementation dependent:

integer code	description
0	normal text
1	inverse video
2	blinking text
3	solid background
4	solid background, inverse video
5	solid background, inverse video, blinking

Suggested values of *color* for text mode operations

Text size can assume the following values:

integer code	description
0	normal text size
1	large text size
2	very large text size
3	small text size

Allowed values of *text size*

The support of text sizes other than the default is optional, and if other sizes are available they will be either those supported in the hardware or text sizes that are chosen by the implementor to give a pleasing balance to text output on the screen.

4.6 Look-up Table I/O

The ITT and VLUT values are both to be loaded and handled as floating point values in the range 0.0 to 1.0. In most devices the VLUT is actually a small integer array with $2^{z_{max}}$ values of 8 or 12 bits each, but handling both the ITT and VLUT in floating point allows common support routines and symmetric handling of the loaded and returned values.

Write_Intensity_Transformation_Table `IILWIT` *extended*

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	ITT number
In	Integer	Start position
In	Integer	Number of entries
In	Real()	Intensity transformation table
Out	Integer	Error indicator

The *intensity transformation table* is a list of floating point numbers in the range 0.0 to 1.0.

ITT number is an integer in the range 0 to n, where n is fixed for any particular display type, and -1 specifies the ITT currently bound to the selected memory (either selected by default or bound to the *memory identifier* by a call to **Select_Memory_Look-up_Table**).

Table entries are indexed starting with 0, and *start position* indicates the first element of a block of entries that are to be written. The value of *start position* must therefore be in the range 0 to $n-1$, where n is the number of entries in the table.

The *number of entries* must be in the range 1 to $n-$*start position*.

Read_Intensity_Transformation_Table `IILRIT` *extended*

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	ITT number
In	Integer	Start position
In	Integer	Number of entries
Out	Real()	Transformation table
Out	Integer	Error indicator

Parameter definitions are identical to those for **Write_Intensity_Transformation_Table**.

Write_Video_Look-up_Table `IILWLT` *basic*

In	Integer	Display identifier
In	Integer	VLUT number
In	Integer	Start position
In	Integer	Number of entries
In	Real()	Video look-up table
Out	Integer	Error indicator

VLUT number is an integer in the range 0 to n, where n is fixed for any particular display type, and -1 specifies the VLUT currently bound to the selected memory.

The *video look-up table* is a list of R, G, and B intensities in the range 0.0 to 1.0. The table data is stored in a $3 \times n$ floating point array.

Table entries are indexed starting with 0, and *start position* indicates the first element of a block of entries that are to be written. The value of *start position* must therefore be in the range 0 to $n-1$, where n is the number of entries in the table.

The *number of entries* must be in the range 1 to $n-$*start position*.

Read_Video_Look-up_Table `IILRLT` *extended*

In	Integer	Display identifier
In	Integer	VLUT number
In	Integer	Start position
In	Integer	Number of entries
Out	Real()	Video look-up table
Out	Integer	Error Indicator

Parameters are identical to those for **Write_Video_Look-up_Table**.

4.7 Zoom and Pan

The following functions affect the *entire* display and not just individual memories, and therefore provide a different function than **Write_Memory_Scroll** and **Write_Memory-_Zoom** routines. Zooming the entire display and panning (or roaming) around the picture is usually performed under the control of an interactive device, but explicit setting and reading by applications programs is provided for devices without attached interactive devices.

Write_Display_Zoom_and_Pan IIZWZP
extended

In	Integer	Display identifier
In	Integer	x offset
In	Integer	y offset
In	Integer	Zoom factor
Out	Integer	Error indicator

When the display is zoomed only part of the display will be visible. The *x* and *y offsets* control which area of the display will be visible by specifying by how many pixels the display should be offset from the center of the screen. The offset is relative to the default position in screen coordinates: positive x and y values shift the memory upwards and to the right.

Read_Display_Zoom_and_Pan IIZRZP
extended

In	Integer	Display identifier
Out	Integer	x offset
Out	Integer	y offset
Out	Integer	Zoom factor
Out	Integer	Error indicator

Read_Display_Zoom_and_Pan returns the current zoom and pan settings for the whole display.

4.8 Cursor Control and Region of Interest

Cursor control functions provide for initialization of cursor interaction, permit the setting of cursor shapes, and support writing and reading back the cursor location. All cursors must have some suitable default characteristic so that a call to **Initialize_Cursor** is not required before a cursor is displayed. A display is assumed to have a number of cursors each of which can be moved independently. The position of any cursor (selected by the *cursor number*) can be read at any time, regardless of whether or not the cursor is visible. To obtain the pixel value at the cursor position a call to **Read_Memory** is required.

Initialize_Cursor IICINC
extended

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	Cursor number
In	Integer	Cursor shape
In	Integer	Cursor color
In	Integer	x cursor position
In	Integer	y cursor position
Out	Integer	Error indicator

The *cursor position* is given in memory coordinates in the specified memory unless the *memory identifier* is -1, in which case the cursor position is in screen coordinates

If *cursor position* is outside the specified memory or off the screen the cursor position is not changed.

The following cursor shapes are defined:

integer code	description
0	(Implementation dependent)
1	Cross hair (full screen)
2	Cross
3	Open cross
4	Square
5	Diamond
6	Circle
7	Diagonal arrow

Allowed values of *cursor shape*

Devices with programmable cursors may use cursor shape codes > 7 to access special cursor shapes.

The *cursor color* is defined as for **Polyline** and **Plot_Text** in a graphics memory except that a value of 0 is implementation dependent. If the specified *cursor shape* or *cursor color* is not available then the shape and color will be implementation dependent.

Set_Cursor_Visibility IICSCV
basic

In	Integer	Display identifier
In	Integer	Cursor number
In	Logical	Visibility
Out	Integer	Error indicator

This function determines whether a cursor position is indicated by some visible mark on the display. If *visibility* is 'true' then the cursor becomes visible.

Read_Cursor_Position IICRCP *basic*

In	Integer	Display identifier
In	Integer	Input memory identifier
In	Integer	Cursor number
Out	Integer	x position
Out	Integer	y position
Out	Integer	Output Memory identifier
Out	Integer	Error indicator

Read_Cursor_Position must be used to return the cursor location. The returned cursor positions are relative to the input memory's origin unless the *input memory identifier* is -1, in which case the position is relative to the display origin and the *output memory identifier* identifies the memory to which the cursor is currently pointing. Otherwise the *output memory identifier* is equal to the *input memory identifier*.

Write_Cursor_Position IICWCP *basic*

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	Cursor number
In	Integer	x position
In	Integer	y position
Out	Integer	Error indicator

The given cursor position is relative to the specified memory's origin unless the *memory identifier* is -1, in which case the position is relative to the display origin. A display may have a number of cursors, each of which can be moved independently. The position of any cursor (selected by the *cursor number*) can be written at any time, regardless of whether or not the cursor is visible.

The following functions are used to define a rectangular region of interest (ROI) on the visible image. These functions support initialization, visibility, and access to positional information in a ROI, but not the interactive manipulation of a ROI. All that is required is to be able to mark two corners of the ROI on the screen. These marks may be crosses, corner markers, or a full rectangular window outline, depending on hardware and implementation characteristics. No masking or windowing capabilities are implied. All conventions for referring to memory identifiers, colors, and coordinates are the same as in the cursor routines.

Initialize_Rectangular_Region_of_Interest IIRINR
extended

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	ROI marker color
In	Integer	Minimum x position
In	Integer	Minimum y position
In	Integer	Maximum x position
In	Integer	Maximum y position
Out	Integer	ROI indentifier
Out	Integer	Error indicator

The ROI positions and color are as specified in **Initialize_Cursor**. Note that the interface is expected to assign an ROI identifier, which is used by the other ROI routines to reference the ROI.

Read_Rectangular_Region_of_Interest IIRRRI
extended

In	Integer	Display identifier
In	Integer	Input memory identifier
In	Integer	ROI indentifier
Out	Integer	Minimum x position
Out	Integer	Minimum y position
Out	Integer	Maximum x position
Out	Integer	Maximum y position
Out	Integer	Output memory identifier
Out	Integer	Error indicator

The *input* and *output memory identifiers* are as specified in **Read_Cursor_Position**.

Write_Rectangular_Region_of_Interest IIRWRI
extended

In	Integer	Display identifier
In	Integer	Memory identifier
In	Integer	ROI identifier
In	Integer	Minimum x position
In	Integer	Minimum y position
In	Integer	Maximum x position
In	Integer	Maximum y position
Out	Integer	Error indicator

Parameters are identical to those for **Write_Cursor_Position**.

Set_Visibility_Rectangular_Region_of_Interest `IIRSRV` *extended*

In	Integer	Display identifier
In	Integer	ROI identifier
In	Logical	Visibility
Out	Integer	Error indicator

This function sets the visibility of the ROI markers on the screen. Parameters are identical to those for **Set_Cursor_Visibility**.

4.9 User Interaction

User interaction functions are provided to support a set of interactive operations (move, zoom, rotate, *etc.*) between some interactive device (evaluator or locator) and some object such as a cursor, color look-up table, or memory. Triggers are provide to signal significant events such as the end of an interaction or marking a position.

Enable_Interaction is used to bind an interactor to an object and, for devices capable of asynchronous operation, to enable the beginning of interaction. More than one interactor/object pair may be active at any given time (*e.g.*, track ball with color table and trigger with memory).

Enable_Interaction `IIIENI` *basic*

In	Integer	Display identifier
In	Integer	Interactor type
In	Integer	Interactor indentifier
In	Integer	Object type
In	Integer	Object identifier
In	Integer	Interactive operation
In	Integer	Exit trigger number
Out	Integer	Error indicator

The *interactor type* selects the different operator interactive devices that are to be used by the user to interact with the object. The following interactor types are defined:

integer code	description
0	Locator
1	Real evaluator
2	Integer evaluator
3	Logical evaluator (switch)
4	Character evaluator
5	Trigger

Allowed values of *interactor type*

Implementation and device-specific types may also be provided in addition to the basic set.

Interactor number selects which actual unit is to be used, since there may be more than one unit of the same type available on a device.

Object type defines the functional unit of the image display system which is to be modified by the interaction. The following types are defined:

integer code	description
0	No visible effect
1	Cursor
2	ITT (intensity transformation table)
3	VLUT (video look-up table)
4	ROI (region of interest)
5	Memory
6	Entire display

Allowed values of *object type*

The *object identifier* specifies the number of the selected object. The exact meaning depends on the *object type*; for example, the *object identifier* is the cursor number if *object type* = 1 ('cursor').

The *interactive operation* specifies the action the selected object is to be performed on. The following operation types are defined:

integer code	description
0	Execute application-specific code
1	Move object, *e.g.*, memory or cursor
2	Rotate object, *e.g.*, color table
3	Increase object zoom, *e.g.*, memory or display
4	Reduce object zoom
5	Set object zoom to normal
6	Blink object, *e.g.*, memory sets
7	Modify object, *e.g.*, region of interest

Allowed values of *interactive operation*

Exit trigger is the number of the trigger used to signal the end of the interactive operation.

The behavior of this routine is determined by the capabilities of the device: on devices capable of handling the interaction without the involvement of the host computer, it starts the interaction on the device and returns. On less sophisticated devices it simply stores the relevant data and exits. In the latter case the interaction only begins when **Execute_Interaction_and_Wait** is called.

The synchronization of completion of an interactive sequence is handled by **Execute_Interaction_and_Wait**. For devices that cannot handle asynchronous operation, all of the manipulation of the interactive session must be handled within this routine. On more powerful devices that can do some of this interaction management in hardware, **Execute_Interaction_and_Wait** may only perform a synchronization function.

Execute_Interaction_and_Wait IIIEIW *basic*

In	Integer	Display identifier
Out	Logical()	Trigger status array
Out	Integer	Error indicator

The routine returns only when an enabled trigger has been fired, or if an interactive operation of type 0 (application-specific interaction) has been requested. In this latter case the application program is allowed to execute some application-specific code, to re-enter the execute interaction routine, and loop until the exit trigger is fired. The actual value of some event caused by a user action, *e.g.*, moving a joystick or entering an input string, must be read by calling the appropriate input function.

The *trigger status array* is an array of logicals indicating which of the triggers on the device have been fired (value set to 'true', or non-zero). The length of this array is known from the device capabilities table, which is accessed via the routine **Query_Capabilities_Integer**. The trigger status array is cleared (all values set to 'false', or zero) on entry to the routine.

Stop_Interactive_Input IIISTI *extended*

In	Integer	Display identifier
Out	Integer	Error indicator

Stop_Interactive_Input cancels all the object/interactor bindings and terminates any interactions that are in progress.

Query_Interactor_Description IIIQID *basic*

In	Integer	Display identifier
In	Integer	Interactor type
In	Integer	Interactor number
Out	String	Interactive device description
Out	Integer	Length of description string
Out	Integer	Error indicator

Query_Interactor_Description returns a text string that contains a description of an interactive device. It allows suitable help messages to be displayed in a device independent way. For example, strings such as "move track ball", "use joystick", "position mouse", "push center button", and so on, could be used to describe the nature of the interactive device so that such prompt messages would not need to be hard-coded in the application.

The maximum allowed length of the *interactive device description* string is 80 characters. The output argument *length of description string* contains the significant length of the string, *i.e.*, with all trailing blanks removed.

The allowed values for *interactor type* are the same as for **Enable_Interaction**.

Get_Locator_Displacement

IIIGLD
extended

In	Integer	Display identifier
In	Integer	Locator number
In	Integer	x displacement
In	Integer	y displacement
Out	Integer	Error indicator

Get_Locator_Displacement returns the displacement of the locator indicated by *locator number*. Locators, such as joysticks or track balls, return the (x, y) displacement from their previous position, rather than an offset from some origin.

Get_Integer_Evaluator

IIIGIE
extended

In	Integer	Display identifier
In	Integer	Evaluator number
Out	Integer	Input integer value
Out	Integer	Error indicator

Get_Integer_Evaluator returns the *input integer value* that was entered on the input device specified by *evaluator number*.

Get_Real_Evaluator

IIIGRE
extended

In	Integer	Display identifier
In	Integer	Evaluator number
Out	Real	Input real value
Out	Integer	Error indicator

Get_Real_Evaluator returns the *input real value* that was entered on the input device specified by *evaluator number*.

Get_Logical_Evaluator

IIIGLE
extended

In	Integer	Display identifier
In	Integer	Evaluator number
Out	Logical	Input logical value
Out	Integer	Error indicator

Get_Logical_Evaluator returns the *input logical value* that was entered on the input device specified by *evaluator number*.

Get_String_Evaluator

IIIGSE
extended

In	Integer	Display identifier
In	Integer	Evaluator number
Out	String	Input string
Out	Integer	Length of returned string
Out	Integer	Error indicator

Get_String_Evaluator returns the *input string* that was entered by the user on the input device specified by *evaluator number* (different *evaluator numbers* might correspond, for example, to a device with multiple keyboards). The maximum allowed length of the *input string* is 80 characters. The output argument *length of returned string* contains the actual length of the string.

4.10 Miscellaneous Functions

Create_Snapshot

IIDSNP
optional

In	Integer	Display identifier
In	Integer	Color mode
In	Integer	Number of pixels
In	Integer	Data position in x
In	Integer	Data position in y
In	Integer	Data depth (bits/pixel)
In	Integer	Packing factor
Out	Integer()	Image data
Out	Integer	Error indicator

Create_Snapshot is used to obtain a snapshot of the picture that appears on the screen, *not* a copy of all of the data that has been loaded into the display. **Create_Snapshot**

returns a pixel representation of either a monochrome image or each of the R, G, and B components for composition into a form suitable for use with a hardcopy device. Data is read out starting at (x, y) in row order, packed as indicated by *data depth* and *packing factor*, and utilizing the currently established transfer window. When *data position in* x or *data position in* y is non-zero, care must be taken to be sure that the transfer window has been set properly. If the image is RGB, then each image is returned separately.

The values for *color mode* are:

integer code	description
0	monochrome/pseudocolor
1	red
2	blue
3	green

Allowed values of *color mode*

The remaining arguments are identical to those of **Read_Memory**.

Blink_Memories

`IIMBLM`
optional

In	Integer	Display identifier
In	Integer()	List of memory identifiers
In	Integer	Number of memory identifiers
In	Real()	Blink periods (in seconds)
Out	Integer	Error indicator

This function sets up the blink periods for the identified memories. The routines **Enable_Interaction** and **Execute_Interaction_and_Wait** are used for interactive control.

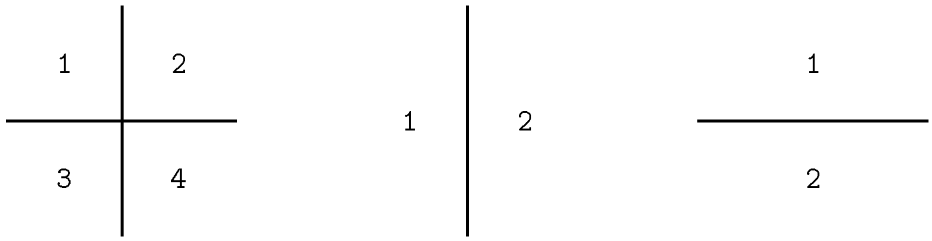
Set_Split_Screen

`IIDSSS`
optional

In	Integer	Display identifier
In	Integer()	Memory identifiers
In	Integer()	x offsets
In	Integer()	y offsets
In	Integer	Split flag
In	Integer	Split x location
In	Integer	Split y location
Out	Integer	Error indicator

Set_Split_Screen allows the screen to be split into up to four segments (numbered 1 – 4).

The form of the split is controlled by *split flag* as follows:



(*split flag* = 0) (*split flag* = 1) (*split flag* = 2)

The index of the memory identifier array determines the screen segment to which each memory ID is bound; *e.g.*, the second memory ID is bound to screen segment 2, the location of which is determined by *split flag*.

Split x location and *split y location* specify the (x, y) position of the intersection of the split screen segments. *Split y location* is irrelevant for *split flag* $= 1$, and *split x location* is irrelevant for *split flag* $= 2$.

The *x* and *y offsets* determine the positioning of the image memory with respect to the lower left corner of the corresponding split screen segment.

A split flag of -1 is used to reset the display with no split and with the first memory visible.

Set_Intensity_Bar_Visibility

`IILSBV`
optional

In	Integer	Display identifier
In	Integer	Memory identifier
In	Logical	Visibility
Out	Integer	Error indicator

Setting the intensity bar *visibility* to 'true' causes an intensity bar to be displayed corresponding to the ITT and VLUT selected for the given *memory identifier*. Making the intensity bar visible may have the effect of making other data invisible depending on the capabilities of the device.

Diagnostic_Routine

`IIDIAG`
optional

In	Integer	Display identifier
In	Integer	Output identifier
Out	Integer	Error indicator

This routine will cause implementation specific diagnostic data to be written to the selected output device. The meaning of the *output identifier* is implementation dependent; it could be, for example, a FORTRAN unit number. *Output identifier* 0 is reserved for ASCII text output to the alphanumeric memory of the image display device. If the output is directed to a text file, it is suggested that the default name for this file be DIAGNOSTIC.IDI.

4.11 Workstation Interface

A limited number of routines are provided to allow the use of a workstation as an image display device. These routines do not attempt to duplicate or replace the windowing environment of a workstation. Rather, they assume that window management is handled outside the interface.

The configuration of memories for a workstation is carried out in the following way: **Enable_Configuration** is called to begin the configuration process. Then, **Allocate_Memory** is called as many times as necessary to define the characteristics of the memories needed. A call to **Stop_Configuration** terminates the configuration process, and then **Select_Configuration** is called in order to make the configuration available for use by the application. It is not necessary to call **Query_Defined_Configuration** since the memory identifiers are already known from calls to **Allocate_Memory**. A configuration which has been dynamically allocated in this manner can be released by calling **Release_Configuration**.

Enable_Configuration `IIDENC`
optional

In	Integer	Display identifier
Out	Integer	Error indicator

Enable_Configuration marks the start of a sequence of configuration operations. It must be used only if additional memory configurations are required and a "negotiation" must be carried out instead.

Allocate_Memory `IIDAMY`
optional

In	Integer	Display identifier
In	Integer	Requested x size
In	Integer	Requested y size
In	Integer	Requested memory depth
In	Integer	Requested memory type
Out	Integer	New memory identifier
Out	Integer	Error indicator

Allocate_Memory attempts to allocate a new memory of the requested type. The *New memory identifier* is returned by the interface, and is used subsequently to reference the newly allocated memories. The *memory depth* is given in bits, and the *memory type* is the same as is defined for **Query_Defined_Configuration**. An error is returned if the memory request cannot be satisfied.

Stop_Configuration

IIDSTC
optional

In	Integer	Display identifier
Out	Integer	Configuration number
Out	Integer	Error indicator

When **Stop_Configuration** is called the display driver software will allocate physical memory to the memories created since the last call to **Enable_Configuration**. By postponing the allocation of physical memory until all the memories required by the application have been defined, the driver software can allocate the available memory in an optimum way. The *configuration identifier* returned is then used in a call to **Select_Configuration** in order to enable the use of the newly defined memories.

Release_Configuration

IIDRLC
optional

In	Integer	Display identifier
In	Integer	Configuration number
Out	Integer	Error indicator

Release_Configuration returns the physical memory allocated to the specified *configuration number* to the pool of unused memory available for the creation of new memories. The *configuration number* and its associated *memory identifiers*, defined with **Allocate_Memory**, become invalid but may be reused when creating new memories in a subsequent negotiation for a new configuration.

4.12 Escape Functions

There are two escape functions which will read and write parameters by name. These may be used to query or set any implementation specific parameters. Use of these will, of course, yield highly implementation and device specific code.

Get_Escape_Parameter

IIEGEP
optional

In	String	Parameter name
In	Integer	Length of output buffer
Out	String	Buffer for string parameter
Out	Integer	Error indicator

Put_Escape_Parameter IIEPEP *optional*

In	String	Parameter name
In	Integer	length of input buffer
In	String	Buffer for string parameter
Out	Integer	Error indicator

A Query_Capabilities names

This table lists the types of image display device capabilities or status values that can be determined via calls to **Query_Capabilities_Integer** or **Query_Capabilities_Real**. The first column of the table contains the type of capability. The integer code for this capability is given in the second column. Programs would use the integer code, either directly, or (preferably) via an include file which defines useful mnemonics for the capabilities. The third column of the table describes the return arguments *output array* and *number of values returned*. An entry under the column 'Output array' of n means that the value is an arbitrary integer; an entry of $n()$ means that there is an array of integer values. Similary, entries of r and $r()$ indicate real-valued return arguments. Entries l and $l()$ are used to represent logical scalar and array values; 'false' is represented as an integer value of 0 and 'true' is represented as any non-zero integer value.

Capability	Integer Code	Output array
implementation level	1	n (0 = basic, 1 = standard)
number of available configurations	10	n
selected configuration number	11	n
physical size of display	12	n_x, n_y
display depth (number of bits in DAC)	13	n
maximum depth of VLUTs	14	n
number of VLUTs	15	n
number of ITTs per image memory	16	n
zoom range	17	n_{min}, n_{max}
memory visible (logical)	20	$l()$
list of memories in currently selected configuration	21	$n()$
depth of memories (in bits)	22	$n()$
list of current VLUT bindings	23	$n()$
list of current display path bindings	24	$n()$
list of current ITT bindings	25	$n()$
maximum dimensions of transfer window in x	30	$x()$
maximum dimensions of transfer window in y	31	$y()$
transfer window x sizes	32	$x()$
transfer window y sizes	33	$y()$
transfer window x offsets	34	$x()$
transfer window y offsets	35	$y()$
depth of transfer windows (in bits)	36	$n()$
number of available device cursors	40	n
array of cursor shapes	41	$n()$
number of cursor shapes	42	n
list of current cursor bindings	43	$n()$
list of current cursor shapes	44	$n()$
list of current cursor colors	45	$n()$
list of current cursor visibilities	46	$l()$

Capability	Integer Code	Output array
number of locators	50	n
number of real evaluators	51	n
number of integer evaluators	52	n
number of logical evaluators (switches)	53	n
number of character evaluators	54	n
number of triggers	55	n
ROI implemented (logical)	60	l
number of device ROIs	61	n
list of current ROI bindings	62	$n()$
list of current ROI marker colors	63	$n()$
list of current ROI visibilities	64	$l()$
blink implemented (logical)	70	l
blink period for each memory (sec)	71	$r()$
split screen implemented (logical)	80	l
split screen x memory offsets	81	$x()$
split screen y memory offsets	81	$y()$
split screen (x, y) address	82	x, y
split screen enabled (logical)	83	l
intensity bar implemented (logical)	90	l
intensity bar visibile (logical)	91	$l()$
snapshot implemented (logical)	100	l
escape function implemented (logical)	101	l
diagnostic routine implemented (logical)	102	l
dynamic configuration implemented (logical)	103	l

B IDI Device Name Standards

The following list contains suggested standards for image display device names. The list is not comprehensive, but rather suggestive of the kind of naming conventions that should be followed. Short (three-character) device names are defined for the convenience of the implementor; long, more readily identifiable names are given for the sake of clarity.

The short names are constructed in the following way: the first two letters are derived from the manufacturer's name and are generally the first and last letters (Gould → GD, Ramtek → RK, *etc.*), except for manufacturers whose names are purely alphabetic (VTE, AED) where we have used the first two letters. The third character of the short name is a mnemonic where possible and sequential (0-based) otherwise. The long names are intended to be as readable as possible but without embedded spaces (which are replaced with underscores). A 'W' is used as the third character of the short name for devices that are workstations.

Device long name	Short name	Notes
GOULD_IP8500l	`GD8`	IP8500, low-resolution
GOULD_IP8500H	`GDH`	IP8500, high-resolution
GOULD_IP8400L	`GD8`	uses IP8500 driver
GOULD_IP9500	`GD9`	IP9500
GOULD_IP9400	`GD9`	uses IP9500 driver
GOULD_FD5000	`GD5`	
IIS_M6700	`IS6`	model 6700
IIS_M70	`IS7`	model 70
IIS_M75	`IS8`	model 75
IIS_IVAS	`ISI`	IVAS
RAMTEK_9200	`RK2`	
RAMTEK_9400	`RK4`	
GRINNELL	`GL0`	
ARGS_7000	`AS7`	
EIDOBRAIN_7001	`EN1`	
VTE	`VT0`	
PARALLAX_640	`PX6`	various μVAX board sets
PARALLAX_1000	`PX1`	
PARALLAX_1280	`PX2`	
AED	`AE0`	
PERITEK	`PK0`	
VICOM	`VM0`	
JUPITER	`JR7`	
COMTAL	`CL0`	
VS_11	`VS1`	
SUN_WS	`SNW`	
MICROVAX_GPX_WS	`MXW`	Ultrix/X Windows
MICROVAX_GPX_WS	`MVW`	VMS version
APOLLO_WS	`AOW`	

C Index of Image Display Interface Subroutines

The table below lists all of the IDI subroutine names, along with their Fortran 77 bindings, C bindings, and their classification (basic, extended, or optional). C programmers may find it more convenient to use the full subroutine names, since these are more obvious in meaning that the short names. However, some C compilers truncate subroutine names to 8 characters, and other subroutines in the interface may lead to name conflicts with other subroutine libraries. Thus, the use of the long names should be done only through the use of `define` statements, and the object libraries should contain references to the short names only.

Subroutine Name	F77 Name	C Name	Class
Open_Display	IIDOPN	IIDOPN_C	basic
Close_Display	IIDCLO	IIDCLO_C	basic
Reset_Display	IIDRST	IIDRST_C	basic
Update_Display	IIDUPD	IIDUPD_C	basic
Get_Error	IIDERR	IIDERR_C	basic
Query_Device_Characteristics	IIDQDV	IIDQDV_C	basic
Query_Capabilities_Integer	IIDQCI	IIDQCI_C	basic
Query_Capabilities_Real	IIDQCR	IIDQCR_C	basic
Query_Defined_Configuration	IIDQDC	IIDQDC_C	basic
Select_Configuration	IIDSEL	IIDSEL_C	basic
Set_Memory_Visibility	IIMSMV	IIMSMV_C	basic
Write_Memory_Scroll	IIZWSC	IIZWSC_C	basic
Write_Memory_Zoom	IIZWZM	IIZWZM_C	basic
Read_Memory_Scroll_and_Zoom	IIZRSZ	IIZRSZ_C	extended
Select_Memory_Look-up_Tables	IIMSLT	IIMSLT_C	extended
Select_Display_Path	IIDSDP	IIDSDP_C	extended
Write_Memory	IIMWMY	IIMWMY_C	basic
Clear_Memory	IIMCMY	IIMCMY_C	basic
Read_Memory	IIMRMY	IIMRMY_C	basic
Set_Transfer_Window	IIMSTW	IIMSTW_C	basic
Polyline	IIGPLY	IIGPLY_C	extended
Plot_Text	IIGTXT	IIGTXT_C	extended
Write_Intensity_Transformation_Table	IILWIT	IILWIT_C	extended
Read_Intensity_Transformation_Table	IILRIT	IILRIT_C	extended
Write_Video_Look-up_Table	IILWLT	IILWLT_C	basic
Read_Video_Look-up_Table	IILRLT	IILRLT_C	extended
Write_Display_Zoom_and_Pan	IIZWZP	IIZWZP_C	extended
Read_Display_Zoom_and_Pan	IIZRZP	IIZRZP_C	extended

white out this line and line below

Set_Visibility_Rectangular_Region_of_Interest	IIRSRV	IIRSRV_C	extended

Subroutine Name	F77 Name	C Name	Class
Initialize_Cursor	IICINC	IICINC_C	extended
Set_Cursor_Visibility	IICSCV	IICSCV_C	basic
Read_Cursor_Position	IICRCP	IICRCP_C	basic
Write_Cursor_Position	IICWCP	IICWCP_C	basic
Initialize_Rectangular_Region_of_Interest	IIRINR	IIRINR_C	extended
Read_Rectangular_Region_of_Interest	IIRRRI	IIRRRI_C	extended
Write_Rectangular_Region_of_Interest	IIRWRI	IIRWRI_C	extended
Set_Visibility_Rectangular_Region_of_Interest	IIRSRV	IIRSRV_C	extended
Enable_Interaction	IIIENI	IIIENI_C	basic
Execute_Interaction_and_Wait	IIIEIW	IIIEIW_C	basic
Stop_Interactive_Input	IIISTI	IIISTI_C	extended
Query_Interactor_Description	IIIQID	IIIQID_C	basic
Get_Locator_Displacement	IIIGLD	IIIGLD_C	extended
Get_Integer_Evaluator	IIIGIE	IIIGIE_C	extended
Get_Real_Evaluator	IIIGRE	IIIGRE_C	extended
Get_Logical_Evaluator	IIIGLE	IIIGLE_C	extended
Get_String_Evaluator	IIIGSE	IIIGSE_C	extended
Create_Snapshot	IIDSNP	IIDSNP_C	optional
Blink_Memories	IIMBLM	IIMBLM_C	optional
Set_Split_Screen	IIDSSS	IIDSSS_C	optional
Set_Intensity_Bar_Visibility	IILSBV	IILSBV_C	optional
Diagnostic_Routine	IIDIAG	IIDIAG_C	optional
Enable_Configuration	IIDENC	IIDENC_C	optional
Allocate_Memory	IIDAMY	IIDAMY_C	optional
Stop_Configuration	IIDSTC	IIDSTC_C	optional
Release_Configuration	IIDRLC	IIDRLC_C	optional
Get_Escape_Parameter	IIEGEP	IIEGEP_C	optional
Put_Escape_Parameter	IIEPEP	IIEPEP_C	optional