SUN/65.8

Starlink Project
Starlink User Note 65.8

Nicholas Eaton

20 July 1995

IDI
Image Display Interface

Programmer's Guide
1.4
Programmers' Manual

Abstract

The Image Display Interface (IDI) is an international standard for displaying astronomical data on an image display. The specification of the interface is given in Terrett et al (1988). This guide is an introduction to programming with IDI, and describes the details of the current implementation. The user should refer to the specification document, for details of the subroutine calls.

Contents

1 Why Use IDI?

IDI is intended for those users (applications) that need to be able to manipulate images on an image display to a greater extent than is available with GKS and its offspring (e.g. PGPLOT). GKS allows an image to be displayed, its look-up table changed and a cursor to be moved over it. The most obvious advantage of IDI over GKS is that it allows the image to be scrolled and zoomed. Other differences that users may wish to exploit is the ability to blink images and to read back a representation of the whole display, which can then be used to obtain a hardcopy of it.

IDI allows these functions to be programmed in a device independent way. Once IDI has been implemented for one device then the same program can run on the different devices just by supplying the appropriate device name to **Open Display**, with the proviso that not all the routines have been implemented on all devices.

This does not mean that IDI supercedes GKS. IDI was written to offer features not available in GKS. It does not have the sophistication of GKS for producing vector (line) plots or character annotation. IDI does have routines to draw lines and plot text, but these are primitive and offer the user little control over how the result will appear, in terms of character sizes, style of line widths etc.

The major strength of IDI is the ability to perform many types of interaction using the mouse. This can be programmed to move the cursor, or the memories, rotate the look-up table, zoom up and down, blink the memories, or perform even more complex tasks by passing control back to the calling program.

It should be noted that it is not possible to mix calls from GKS and IDI on the same display; the two packages use completely different models of the display. An application could, however, use the packages one after the other by utilising the Applications Graphics Interface (SUN/48) to mediate between them.

2 Summary of IDI calls

The following list gives all the top-level routines in the interface that have been implement by Starlink An error status will be returned if a routine that has not been implemented is called. The routine arguments are defined in the specification document.

Control

IIDOPN(devnam, dispid, status)
    Open Display

IIDCLO(dispid, status)
    Close Display

IIDRST(dispid, status)
    Reset Display

IIDUPD(dispid, status)
  Update Display

IIDERR(status, messag, meslen)
  Get Error

Configuration

IIDQDV(dispid, nconf, xsize, ysize, depth, nvlut, nitt, ncurs, status)
  Query Device Characteristics

IIDQCI(dispid, capid, narr, outarr, nout, status)
  Query Capabilities Integer

IIDQCR(dispid, capid, narr, outarr, nout, status)
  Query Capabilities Real

IIDQDC(dispid, nconf, memtyp, nmemax, modcon, memid, memsix, memsiy, memdep, ittdep, nmem, status)
  Query Defined Configuration

IIDSEL(dispid, nconf, status)
  Select Configuration

Memories

IIMSMV(dispid, memid, nmem, lvis, status)
  Set Memory Visibility

IIZWSC(dispid, memid, nmem, xoff, yoff, status)
  Write Memory Scroll

IIZWZM(dispid, memid, nmem, zoomf, status)
  Write Memory Zoom

IIZRSZ(dispid, memid, xoff, yoff, zoomf, status)
  Read Memory Scroll and Zoom

IIMSLT(dispid, memid, lutnum, ittnum, status)
  Select Memory Look-up Tables

IIMWMY(dispid, memid, image, npix, depth, pack, xstart, ystart, status)
  Write Memory

IIMCMY(dispid, memid, nmem, back, status)
  Clear Memory

IIMRMY(dispid, memid, npix, xstart, ystart, depth, pack, itton, image, status)
  Read Memory

IIMSTW(dispid, memid, direcn, xsize, ysize, depth, xoff, yoff, status)
  Set Transfer Window

Graphics

IIGPLY(dispid, memid, x, y, nxy, color, lstyle, status)
Polyline

IIGTXT(dispid, memid, text, xpos, ypos, tpath, tangle, color, tsize, status)
Plot Text

Look-up Table

IILWIT(dispid, memid, ittnum, start, nent, itt, status)
Write Intensity Transformation Table

IILRIT(dispid, memid, ittnum, start, nent, itt, status)
Read Intensity Transformation Table

IILWLT(dispid, lutnum, start, nent, vlut, status)
Write Video Look-up Table

IILRLT(dispid, lutnum, start, nent, vlut, status)
Read Video Look-up Table

Zoom and Pan

IIZWZP(dispid, xoff, yoff, zoomf, status)
Write Display Zoom and Pan

IIZRZP(dispid, xoff, yoff, zoomf, status)
Read Display Zoom and Pan

Cursor

IICINC(dispid, memid, numcur, shape, color, xc, yc, status)
Initialize Cursor

IICSCV(dispid, numcur, lvis, status)
Set Cursor Visibility

IICRCP(dispid, inmid, numcur, xc, yc, outmid, status)
Read Cursor Position

IICWCP(dispid, memid, numcur, xc, yc, status)
Write Cursor Position

Region of Interest

IIRINR(dispid, memid, roicol, xmin, ymin, xmax, ymax, roiid, status)
Initialize Rectangular Region of Interest

IIRSRV(dispid, roiid, lvis, status)
Set Visibility Rectangular Region of Interest

IIRRRI(dispid, inmid, roiid, xmin, ymin, xmax, ymax, outmid, status)
Read Rectangular Region of Interest

IIRWRI(dispid, memid, roiid, xmin, ymin, xmax, ymax, status)
Write Rectangular Region of Interest

Interaction

IIIENI(dispid, intty, intid, objty, objid, intop, extrn, status)
Enable Interaction

IIIEIW(dispid, trigs, status)
Execute Interaction and Wait

IIISTI(dispid, status)
Stop Interactive Input

IIIQID(dispid, intty, intid, messag, meslen, status)
Query Interactor Description

IIIGLD(dispid, locnum, dx. dy, status)
Get Locator Displacement

Miscellaneous routines

IIDSNP(dispid, cmode, npix, xstart, ystart, depth, pack, image, status)
Create Snapshot

IILSBV(dispid, memid, lvis, status)
Set Intensity Bar Visibility

Workstation interface

IIDENC(dispid, status)
Enable Configuration

IIDAMY(dispid, xsize, ysize, depth, memtyp, memid, status)
Allocate Memory

IIDSTC(dispid, nconf, status)
Stop Configuration

IIDRLC(dispid, nconf, status)
Release Configuration

ADAM interface routines

IDI_ANNUL(dispid, status)
: Annul display in the ADAM environment

IDI_ASSOC(pname, acmode, dispid, status)
: Associate display in the ADAM environment

IDI_CANCL(pname, status)
: Cancel display parameter in the ADAM environment

3 Using the IDI Library

3.1 Control

3.1.1 Opening and Closing IDI

All programs using IDI to perform graphics I/O on a device have to begin and end the IDI sections with calls to **Open Display** and **Close Display**. If the program is an ADAM task then these two calls have to be replaced by appropriate calls to IDI_ASSOC and IDI_CANCL or IDI_ANNUL (see Appendix B for more details on these functions).

More than one device can be open at a time, each needing separate calls to **Open Display** and **Close Display**. The devices are selected by the unique identifier returned from **Open Display**, and these identifiers should not be tampered with. Device selection is achieved by passing the relevant device identifier to subsequent IDI routines.

By default IDI does not reset the display when the device is opened. This enables applications using IDI to manipulate pictures drawn by other applications. If an application requires the device to be in its default state then an explicit call to **Reset Display** is required.

As an example consider an application that is drawing a picture from afresh, and therefore wants the device reset at the beginning. If the application is an ADAM task IDI is opened using

```
CALL IDI_ASSOC( 'DEVICE', 'WRITE', ID, STATUS )
```

where 'DEVICE' is the name of a parameter in the interface file through which the name of the device is obtained. The access mode is set to 'WRITE' which forces a reset of the device, the other possibilities 'READ' and 'UPDATE' leave the device as it was found. The third argument is the display identifier which is returned from the routine, and is used in subsequent IDI calls to direct operations to this device. The status argument is an integer which indicates if any problems have occurred.

In a non-ADAM application the same effect is achieved with

```
CALL IIDOPN( DNAME, ID, STATUS )
CALL IIDRST( ID, STATUS )
```

where DNAME is a string containing the name of the device, corresponding to an entry in the file gns_idinames. The display identifier returned from **Open Display** is passed to **Reset Display** to indicate that this is the device to reset.

3.1.2 Buffering

For efficiency IDI can save up its output and send it to the graphics device in batches. There may be occasions, therefore, when the device does not have the complete picture displayed. To force the program to flush the output buffer a call to **Update Display** is required:

```
            CALL IIDUPD( ID, STATUS )
```

Quite often IDI will automatically flush the output buffer, for example when the buffer is full, when the program is reading from the display, or when using the mouse. The output buffer is also flushed when IDI is closed down so that the picture is guaranteed to be complete after a call to **Close Display**.

3.1.3 Error messages

IDI returns error status values when it cannot complete a requested function satisfactorily. These status values can be converted into error messages by calling **Get Error**:

```
            CALL IIDERR( STATUS, MESSAG, MESLEN )
```

The first argument is the integer status value returned from an IDI routine, and the string containing the error message is returned in the second argument. If the status value is not defined then a blank string is returned from this routine.

IDI does not use the usual Starlink concept of inherited status. The status value passed to each routine is initialised to IDI__OK (= 0) at the beginning of each routine. This puts the onus on the applications programmer to provide sufficient status checks to locate the source of any problem. If an application is using an inherited status scheme it is probably best to pass a local status variable to the IDI routines, and if an error occurs copy this to the inherited status of the calling routine, or deal with the error in some other way.

3.2 Memories

IDI uses memories as its drawing board. The memories can be considered to be rectangular areas of pixels held internally in the display device. The manner in which these memories appear on the screen is also controlled by IDI. An image is displayed on a device by loading the pixel values into a memory. The pixel values are an array of unsigned integers in the range 0 to $2^{depth} - 1$ where *depth* is the memory depth (bits per pixel). Any values that exceed the maximum are truncated by discarding the most significant bits. The memory can be displayed at different positions or with different zoom factors, and the colour representation of the image can be altered by changing the display path (see below).

Images are loaded into a memory using **Write Memory** and they can be read back from a memory using **Read Memory**. Images do not have to fill the whole memory and the selection of a section of memory to write to is achieved using **Set Transfer Window**. A transfer window is a rectangular sub-section of memory and the process of writing or reading an image employs the transfer window to define which part of the memory is to be used. If required the image loading can begin in the middle of a row. A memory is cleared by filling the memory with a constant value (usually zero) using the routine **Clear Memory**.

Writing an image into memory with **Write Memory** does not guarantee that the image will appear on the display. Although this may happen with some devices it should not be relied upon, and a call to **Set Memory Visibility** should be used to ensure consistent behaviour.

Suppose an application wants to display an image, having NX and NY pixels along each axis, in the bottom left corner of the memory. The application has already checked that the memory is big enough to accept the image, but the size and shape of the image may not match the size and shape of the memory. A transfer window is therefore set up to match the size of the image and located at the bottom left of the memory.

```
      DIRECN = 0
      XOFF = 0
      YOFF = 0
      CALL IIMSTW( ID, MEMID, DIRECN, NX, NY, DEPTH, XOFF, YOFF, STATUS )
```

The memory identifier (MEMID) is used to select one memory from those available on the current configuration. A list of available memory identifiers can be obtained from **Query Defined Configuration** as shown below. The load direction (DIRECN=0) indicates that the image is to be loaded from bottom to top. In this case the size of the transfer window is made to match the size of the image (NX, NY). The X and Y offsets (XOFF, YOFF) indicate where the origin of the transfer window is with respect to the origin of the memory. The depth argument (DEPTH) specifies the expected number of significant bits for each pixel in the input data stream (e.g. data in the range 0 to 255 is stored in eight bits). If this does not exactly match the depth of the memory then data truncation or padding will occur.

The image is injected into the pixel memory using **Write Memory**.

```
      XSTART = 0
      YSTART = 0
      NPIX = NX * NY
      CALL IIMWMY( ID, MEMID, PIXDAT, NPIX, DEPTH, PACK,
     :             XSTART, YSTART, STATUS )
```

The pixel values are stored in the integer input data array (PIXDAT) as a contiguous stream, and the number of pixels to draw is given by NPIX. The data depth (DEPTH) and packing factor (PACK) indicate how individual pixel values are stored in the integer elements of the input array. For instance if the integer word is 32 bits long and the pixel values only have 8 significant bits, then it is possible to store four pixel values in each integer (DEPTH=8, PACK=4). Although this type of packing will save space it will probably not improve plotting times, and so there is no advantage in programs changing the packing of the data themselves. Of course if the data is stored externally in packed form then this feature enables the data to be displayed without having to unpack it first. The start position (XSTART, YSTART) indicates where, with respect to the origin of the transfer window, the first pixel in the data stream is to appear. Subsequent pixels are placed to the right of this one until the edge of the transfer window is reached when a new row of pixels is started. If the size of the transfer window does not match the size of the image then the line breaks will not appear in the correct place and the image will appear skewed.

The final stage of displaying the image is to make the memory visible. This is done with the routine **Set Memory Visibility**.

```
      NMEM = 1
      VIS = 1
      CALL IIMSMV( ID, MEMID, NMEM, VIS, STATUS )
```

This routine takes an array of memory identifiers as its input, so that more than one memory can be made visible at once, but this example only uses the one memory. The visibility (VIS) is a logical value which takes the value TRUE to make the memory visible, and FALSE to make the memory invisible. The IDI specification defines these logical values to be integers having the value zero (FALSE) and non-zero (TRUE).

The display path essentially chooses which intensity transformation table and which look-up table (discussed below) will be used to display the image. The intensity transformation table and look-up table converts the integer values stored in the memory into colours on the screen. If more than one of these tables is available then the bindings of the tables to the memories can be changed using **Select Memory Look-up Tables**.

The position and zoom factor of the memory on the screen can be altered using the routines **Write Memory Scroll** and **Write Memory Zoom**. These routines are non-interactive and the amount of scroll and zoom is set by the input arguments. A zoom factor of 0 means no zoom, a zoom factor of 1 means zoom once (double the size) etc. Negative zooms may be supported on some devices and a zoom factor of -1 means unzoom once (half the size) etc. Scroll offsets are given in pixel units with positive values moving the image to the right and up. Like **Set Memory Visibility** these two routines take a list of memories as their input so that more than one memory can be operated on with one call, if required. A program can inquire the current scroll and zoom settings with **Read Memory Scroll and Zoom**. This routine can be used after an interactive operation to inquire what the final values of any scroll and zoom are.

Two memories can be blinked (alternately displayed) using the routine **Blink Memories**. When this routine is called the blinking begins immediately and continues until the right hand mouse button is pressed. The speed of the blink is initially set by the input argument to the routine, but can be subsequently changed using the left and centre mouse buttons. See the section in the Ikon implementation notes for more details.

3.3 Configuration

The device configuration defines the various capabilities of the device, such as the number of memories, their sizes and their depths, the number of look-up tables etc. A device may have more than one configuration, for example a device with a fixed address space may have one configuration having one memory which fills the available space and another configuration that has several smaller memories allocating the space between them. There are a number of inquiry routines that allow the program to find out various parameters of a given configuration. The inquiries can be used to find out about configurations other than the current one to see if one may be more suitable for the program's requirement. **Query Defined Configuration** gives details of the requested configuration.

If there is more than one configuration available on a device then the routine **Select Configuration** will implement the chosen one.

Query Device Characteristics gives details of the basic set up of the device, such as the size of the display in pixels, the number of configurations available, the number of look-up tables and the number of cursors. More detailed inquiries of the capabilities and current settings for the device can be obtained with **Query Capabilities Integer** and **Query Capabilities Real**. Appendix A gives details of the available capabilities. Note that some of the capabilities are arrays of numbers and so the array arguments should be dimensioned large enough to accommodate the returns, otherwise the information returned will be incomplete. A global variable IDI__MAXCP has been supplied which can be used to dimension the array and is as large as any capability on any implemented device. This is defined in the include file IDI_PAR. Examples of using **Query Capabilities Integer** can be found in the in appendix C.

As an example of searching for a suitable configuration consider an application that requires a minimum size (MEMINX, MEMINY) for an image memory. The first call is to **Query Device Characteristics** to establish, amongst other things, the number of configurations available which is returned in the second argument.

```
      CALL IIDQDV( ID, NUMCON, DXSIZE, DYSIZE, DDEPTH, NLUT, NITT,
     :             NCURS, STATUS )
```

As before the first argument is the display identifier returned from **Open Display**. The other arguments return the display size and depth, the number of look-up tables, the number of intensity transformation tables and the number of cursors.

The application then examines each configuration in turn using **Query Defined Configuration** to establish the size of the memories, returned in the seventh and eighth arguments.

```
*   Want information on image memories, so memory type = 1
      MEMTYP = 1

*   Examine each configuration in turn, starting from configuration 0
      NCONF = 0
  10  CONTINUE
      IF ( NCONF .LT. NUMCON ) THEN
         CALL IIDQDC( ID, NCONF, MEMTYP, IMAXCP, MODCON, MEMIDS,
     :                MEMSIX, MEMSIY, MEMDEP, ITTDEP, NMEM, STATUS )

*   Examine each memory in turn
         M = 0
  20     CONTINUE
         IF ( M .LT. NMEM ) THEN
            M = M + 1

*   Check the memory size against the required one
            IF ( ( MEMSIX( M ) .GE. MEMINX ) .AND.
     :           ( MEMSIY( M ) .GE. MEMINY ) ) GOTO 30
            GOTO 20
         ENDIF

*   Increment the configuration loop counter
         NCONF = NCONF + 1
         GOTO 10
      ENDIF

*   If this point is reached then have not found a suitable memory
*   so abort the program
      GOTO 99

*   A suitable memory has been found with configuration number NCONF
  30  CONTINUE
```

The memory type argument (MEMTYP) is used to indicate which type of memory the routine **Query Defined Configuration** will return information on. The choices are image memories (1), text memories (2) or graphics memories (4).

As with most IDI identifiers the configuration identifiers (NCONF) start from zero and go up to one less than the number of configurations (NUMCON-1).

The information is returned from **Query Defined Configuration** in integer arrays having one entry per memory in the configuration. The size of the arrays can be dimensioned using the global variable (IDI__MAXCP) since memory lists form part of the capability inquiries. The actual number of memories in the configuration is given by the output argument (NMEM). Each memory is examined in turn to see if it fulfills the size criterion.

The output list of memory identifiers (MEMIDS) contains the values to be used to identify a particular memory for subsequent IDI routines, in the way that the display identifier selects a particular device. The other arguments returned from **Query Defined Configuration** are the memory depths, the intensity transformation table depths and the configuration mode (MODCON) which indicates if the device is set up for monochrome (0), pseudo-colour (1) or true-colour (2) display.

The final stage of the example is to remember which memory was chosen and to load the appropriate configuration into the device.

```
*   A suitable memory has been found with configuration number NCONF
  30  CONTINUE

*   Obtain the memory identifier of the chosen memory
      MEMID = MEMIDS( M )

*   Select the appropriate configuration
      CALL IIDSEL( ID, NCONF, STATUS )
```

3.4 Graphics

IDI offers two primitive graphics functions to allow for annotation of an image. The routine **Polyline** will draw a line connecting a sequence of given points. The routine **Plot Text** will draw text, in one of up to four sizes, at a given position and at a given angle. Depending on the capabilities of a device either of these routines may overwrite the pixel values of the given memories, and any existing data at these locations will be lost. The appearance of the text (font style and size in pixels) can differ from device to device, because the IDI specification does not define its own fonts and allows implementors to make use of any hardware facilities. The call to **Plot Text** is as follows:

```
      CALL IIGTXT( ID, MEMID, STRING, XPOS, YPOS, TPATH, TANGLE,
     :             COLOR, TSIZE, STATUS )
```

The text to be drawn is passed in the third argument (STRING). The position of the text is given by the next two arguments (XPOS, YPOS) which define the offset of the bottom left corner of the first character with respect to the memory origin. The text path (TPATH) defines if the text is to run horizontally, vertically or back to front, and the orientation of the string (TANGLE) turns the string the given number of degrees clockwise. Some implementations may only offer the default values (0) for both these arguments with the result that the text will only appear horizontally running from left to right. Depending on the type of memory the colour argument (COLOR) is either one of the set of predefined values or is an index into the current look-up table. The text size (TSIZE) is defined imprecisely and takes one of four values, normal (0), large (1), very large (2) or small (3). The values other than the default (0) may not be implemented.

The routine **Create Snapshot** will return a picture of the display screen in an integer array. It differs from **Read Memory** in that it does not just return the pixel values from one memory, but takes account of all memories visible on the screen, and at each pixel location returns the value from the memory which is visible at that point. It works on the principle of what you see is what you get, and thus the look-up tables currently displayed are used to transform the pixel values into a value representing the colour. For pseudo-colour devices the displayed colour is transformed onto a single scale using the following fractions of each primary colour $0.30 * \mathit{Red} + 0.59 * \mathit{Green} + 0.11 * \mathit{Blue}$. The call to **Create Snapshot** is as follows:

```
      CALL IIDSNP( ID, CMODE, NPIX, XSTART, YSTART, DEPTH, PACK,
     :             IMDAT, STATUS )
```

The colour mode (CMODE) is zero for a pseudo-colour or monochrome device. The position arguments (XSTART, YSTART) define the start position with respect to the origin of the current transfer window. The given number of pixels (NPIX) are read out sequentially from the current transfer window, with line breaks occurring at the right hand edge of the window. The data depth (DEPTH) and packing factor (PACK) indicate how the pixel values are to be stored in the integer elements of the output array (IMDAT).

3.5 Look-up Table

A pixel value is converted into a spot of colour on the screen in two stages. The pixel value is used as an index into the intensity transformation array. This table has one entry for each possible pixel value, with the pixel values in the range 0 to $2^{depth} - 1$ where *depth* is the memory depth (bits per pixel). Each entry in this table contains an index into another array, the look-up table, which gives the proportion of red, green and blue used to make the spot of colour. If the number of entries in the look-up table matches the number of entries in the intensity transformation table, then the default intensity transformation table contains a linear mapping, which means that the pixel values point directly into the look-up table. If the tables are of different length, then the default mapping will scale linearly between the two.

The default look-up table is a linear grey scale, but this can be changed by sending a new look-up table with **Write Video Look-up Table**. To preserve device independence the look-up table entries are stored in a real array with values ranging from 0.0 to 1.0, where the latter indicates the maximum intensity on the device. As an example consider changing one of the screen colours to magenta. The look-up table array is most conveniently defined as a 2-dimensional real array having 3 elements in its first dimension and the number of colour entries in the second.

```
*   Dimension the array of colours, in this example length = 1
      REAL VLUT( 3, 1 )

*   Define the number of entries and the red, green and blue intensities
      NENT = 1
      VLUT( 1, NENT ) = 1.0
      VLUT( 2, NENT ) = 0.0
      VLUT( 3, NENT ) = 1.0

*   Load the colour into the look-up table
      CALL IILWLT( ID, LUTNUM, START, NENT, VLUT, STATUS )
```

The look-up table number (LUTNUM) defines which of the available look-up tables to use. The number of available look-up tables can be inquired using **Query Device Characteristics** or

Query Capabilities Integer. The look-up table is updated starting at the entry defined by the third argument (START), which in this example corresponds to the index of the one colour to change, and continues until the given number of colours (NENT) has been loaded.

Any memory that uses this look-up table will be updated when the new look-up table is sent. The bindings of memories to look-up tables can be changed with **Select Memory Look-up Tables**.

The look-up tables can be read back using **Read Video Look-up Table**.

3.6 Zoom and Pan

All the memories on the screen can be zoomed and panned together using **Write Display Zoom and Pan**. This saves the effort of having to set the zoom and scrolls for all the memories individually using **Write Memory Scroll** and **Write Memory Zoom**. These display settings can be read back using **Read Display Zoom and Pan**. A zoom factor of 0 means no zoom, a zoom factor of 1 means zoom once (double the size) etc. Negative zooms may be supported on some devices and a zoom factor of -1 means unzoom once (half the size) etc. Scroll offsets are given in pixel units with positive values moving the image to the right and up.

3.7 Interaction

The interactions possible with IDI give the user the greatest scope for making use of the image display in an interactive application. The device which controls the interactions is known as the *interactor* and corresponds to a mouse or trackerball, or some similar object. An interactor such as a mouse or trackerball is considered to have two components; the part which controls the two-dimensional movement is known as the *locator*, and the buttons are known as *triggers*.

All interactions are controlled by two basic routines. **Enable Interaction** sets up one interaction per call. If more than one interaction is required then this routine has to be called again with the appropriate arguments. The routine **Execute Interaction and Wait** executes all the current interactions and continues until an exit trigger is pressed. An exit trigger is one of the interactor buttons that is defined in **Enable Interaction** to stop that particular interaction. The exit trigger is therefore a stop button which disables the interaction and returns control to the program. If more than one interaction is defined, for example scrolling and zooming the memory, then it is usual to define the same trigger to be the exit trigger for all interactions.

It is possible to set up more than one interaction to be simultaneously using the same interactor component, for example the locator (mouse) could be used to scroll the memory and rotate the look-up table at the same time, although the effect would be pretty nauseous.

The interactions are defined by two components, the first defines the action to be done, such as move or zoom, and the second defines the object that this action is to be done to, such as cursor or memory. The interaction cross reference table (see Appendix D) shows which actions on which objects are possible for a given device. Clearly some of the possibilities are meaningless, such as rotate display or zoom look-up table, and any attempt to perform such interactions will result in an error.

Examples of setting up interactions are shown in the following section on regions of interest and in the example program (appendix C).

The routine **Stop Interactive Input** will clear out all current interactions previously set up with Enable Interaction. If more than one interactive session is required in a single application, for example getting a user to change the shape of a region of interest, and then getting the user to move the fixed region of interest around the screen, then **Stop Interactive Input** should be called between each.

Messages indicating what the user has to do to manage an interaction can be constructed from the output of the routine **Query Interactor Description**. This will return a string that describes how the given interaction is to be performed, such as 'move mouse', or 'press centre button'. This can be output by the program to give instructions to the user on how to achieve the particular interaction. Examples of the use of this routine can be found in the example program in appendix C.

More complex interactions than those already offered can be programmed using the *application specific code* mode. Normally the program control would remain inside **Execute Interaction and Wait** until an exit trigger is fired. In the *application specific* mode the routine **Execute Interaction and Wait** returns control to the calling program, where other tasks can be performed. The program then calls the routine **Execute Interaction and Wait** again, and this loop is repeated until an exit trigger is fired. The routine **Get Locator Displacement** can be used inside the loop to inquire if the locator has moved, and thus control the programmed interaction. A skeleton example of such an interaction follows.

```
*   Set up an application specific interaction to read the locator position
*   Set up the mouse ( interactor type = 0, interactor id = 0 ) to have
*   no visible effect ( object type = 0, object id = 0 ) on application
*   specific code ( interactive operation = 0 ). End the interaction by
*   pressing the right hand button ( exit trigger number = 2 ).
      INTTY = 0
      INTID = 0
      OBJTY = 0
      OBJID = 0
      INTOP = 0
      EXTRN = 2
      CALL IIIENI( ID, INTTY, INTID, OBJTY, OBJID, INTOP, EXTRN,
     :             STATUS )

*   Loop ( in a non-FORTRAN77 way ) until the exit trigger is fired
      DO WHILE ( TRIGS( EXTRN + 1 ) .EQ. 0 )

*   Enable the interaction
         CALL IIIEIW( ID, TRIGS, STATUS )

*   Inquire the locator displacement
         CALL IIIGLD( ID, INTID, DX, DY, STATUS )

*   Do some application specific functions
         ...

      ENDDO
```

Note in the loop test the exit trigger number is incremented because the trigger numbers start from zero whereas, by default, the FORTRAN array TRIGS has one as its first index. Put another way the exit trigger number of two corresponds to the third entry in the TRIGS array. The displacements returned from **Get Locator Displacement** give the shift in the locator position (in pixels) since the last call to this routine, and these can then be used in the application specific section.

The routine **Execute Interaction and Wait** requires one of its arguments to be a logical array whose length matches the number of triggers on the device. The number of triggers can be obtained via the routine **Query Capabilities Integer**, but standard FORTRAN 77 does not allow dynamic length arrays to be defined. Thus a general purpose application needs to know what is the maximum number of triggers it will have to cope with so it can define the array accordingly. A global parameter IDI__MAXTR has therefore been supplied which can be used to dimension the array, and this is defined in the include file IDI_PAR. The following code segment shows how to define the trigger array in a FORTRAN program

```
*  Global Constants:
      INCLUDE 'IDI_PAR'
*  Local variables:
      INTEGER TRIGS( IDI__MAXTR )
```

3.8 Cursor

The cursor is usually controlled by the interactor (mouse) but there are several routines to give basic cursor control from a program. **Set Cursor Visibility** switches the cursor on or off and **Write Cursor Position** repositions the cursor on the display. The cursor position can be read back using the routine **Read Cursor Position**.

The routine **Initialize Cursor** can be used to change the shape of the cursor to one of the predefined shapes. If more than one cursor colour is supported then this can also be changed using **Initialize Cursor**.

3.9 Region of Interest

A region of interest is a rectangular area on the screen indicated by a rubber-band box. In most respects it is similar in action to a cursor, but instead of indicating a point it defines an area. The routine **Initialize Rectangular Region of Interest** is required to set up a region of interest, and this returns an identifier which is used by the other routines to access the region. **Set Visibility Rectangular Region of Interest** switches the box on or off and **Write Rectangular Region of Interest** repositions the box on the display. The position of the region can be read back using the routine **Read Rectangular Region of Interest**

A region of interest can also be controlled using the interactor (mouse). The *object identifier* field of **Enable Interaction** should contain the relevant ROI identifier returned by **Initialize Rectangular Region of Interest**. A region can either be moved using an *interactive operation* = 1, or its shape can be changed using an *interactive operation* = 7. When moving a region interactively, the size of the area is fixed and the interaction moves the box over the display.

When modifying a region of interest interactively, a rubber-band box is displayed which has one corner anchored and the other corner under the control of the mouse. The active corner is indicated by a small crosshair cursor. By moving the active corner the size and shape of the box is changed. To allow complete control a trigger is set up which switches the active corner to the opposite side. Any region can then be defined by first moving the locator to position one corner as required, switching the active corner, and then moving the locator again to position the opposite corner. This type of operation requires two interactions to be set up. The first defines that the mouse is to move the active corner of the rubber-band box, the second defines a button which will be used to switch the active corner from its current position to the opposite corner of the rubber-band box. The following code segment shows how this works.

```
*   Set up the mouse ( interactor type = 0, interactor id = 0 ) to
*   control the ROI ( object type = 4, object id = ROIID ) by modifying
*   it ( interactive operation = 7 ). End the interaction by pressing
*   the right hand button ( exit trigger number = 2 ).
      INTTY = 0
      INTID = 0
      OBJTY = 4
      OBJID = ROIID
      INTOP = 7
      EXTRN = 2
      CALL IIIENI( ID, INTTY, INTID, OBJTY, OBJID, INTOP, EXTRN,
     :             STATUS )

*   Define the left-hand trigger ( interactor type = 5, interactor id = 0 )
*   to toggle the ROI active corner ( object type = 4, object id = ROIID )
*   while modifying it ( interactive operation = 7 ). End the interaction
*   by pressing the right hand button ( exit trigger number = 2 ).
      INTTY = 5
      INTID = 0
      OBJTY = 4
      OBJID = ROIID
      INTOP = 7
      EXTRN = 2
      CALL IIIENI( ID, INTTY, INTID, OBJTY, OBJID, INTOP, EXTRN,
     :             STATUS )

*   Enable the interaction
      CALL IIIEIW( ID, TRIGS, STATUS )

*   Stop the interaction
      CALL IIISTI( ID, STATUS )
```

3.10 Workstation Interface

When using a workstation as an image display device it is possible to dynamically allocate memory. This is useful if, for instance, the available memory is not large enough to accommodate an image. Consider the earlier example in which the existing configuration was searched for a memory of a minimum size (MEMINX, MEMINY). In that case the program aborted if a suitable memory could not be found. Using the workstation interface a suitably sized memory can be requested. An application should first inquire if the workstation interface is supported using **Query Capabilities Integer**

```
*   Inquire if the workstation interface is supported
      NARR = 1
      CALL IIDQCI( ID, ISDYNC, NARR, YESNO, NVAL, STATUS )

*   Abort if it is not supported
      IF ( YESNO .EQ. 0 ) GOTO 99
```

The global parameter ISDYNC defines the capability number used to inquire if the dynamic configuration is implemented (see appendix A).

The memory allocation is begun with a call to **Enable Configuration** which initialises the process. The memory requirements are passed to **Allocate Memory** and then the process is ended with a call to **Stop Configuration** which returns a *configuration number* to be used to access the new memory. The new configuration has to be selected with **Select Configuration** before the new memory can be used. The following code section shows this in action.

```
*   Begin the configuration process
      CALL IIDENC( ID, STATUS )

*   Request an image memory (MEMTYP=1) of the required size
      MEMTYP = 1
      CALL IIDAMY( ID, MEMINX, MEMINY, DDEPTH, MEMTYP, IMEMID, STATUS )

*   Check that the request has been satisfied
      IF ( STATUS .NE. IDI__OK ) GOTO 99

*   End the configuration process
      CALL IIDSTC( ID, ICONF, STATUS )

*   Select this configuration
      CALL IIDSEL( ID, ICONF, STATUS )
```

4 Linking with IDI

If any of the idi include files are required, create a soft link to them with the command

```
% idi_dev
```

and use an include statement of the form:

```
      INCLUDE 'IDI_PAR'
```

A standalone program can be linked by specifying ‘idi_link‘ on the compiler command line. Thus to compile and link a standalone application called ‘prog’ the following could be used

```
% f77 prog.f -o prog `idi_link`
```

(note the use of the backward quotes).

For programs in the ADAM environment the compile and link command is

```
$ alink prog.f `idi_link_adam`
```

5 Device Names

The choice of display device used by an IDI program is selected by passing the device name to **Open Display** or by defining the name through the parameter system if IDI_ASSOC is used in an ADAM task.

IDI uses the Graphics Name Service (SUN/57) to translate the given device names. Acceptable device names can be found by running the example program `gnsrun_idi` (normally found in `/star/bin/examples`).

6 Errors

The IDI routines return a non-zero status value if the routine was unable to perform its task properly; a successful operation will return a zero value. Note that the standard IDI routines (those beginning with the two character prefix II) do not use inherited status and reset the value of the status argument to IDI__OK (= 0) at the beginning of each routine. Therefore the status value should be frequently checked if the application is to trap any errors.

The routine IIDERR can be used to obtain an error message from an IDI status value. The FORTRAN error definitions are declared in the file IDI_ERR.FOR. The C error definitions are declared in the file idi_err.htxt.

7 Acknowledgements

The X-windows IDI driver is largely based on software kindly supplied by Paolo Santin of the Trieste Astronomical Observatory.

References

[1] D.L.Terrett, P.M.B.Shames, R.J.Hanisch, R.Albrecht, K.Banse, F.Pasian, M.Pucillo and P.Santin., 1988, An image display interface for astronomical image processing, Astron.Astrophys.Suppl.Ser., **76**, 263-304.

A Query capabilities names

The table gives the types of capabilities that can be determined via calls to **Query Capabilities Integer** or **Query Capabilities Real**. The first column lists the capability. The second column gives the integer code to be used in the call. Alternatively by including the FORTRAN file 'IDI_PAR' in a program the codes can be accessed using the mnemonics given in the third column. The final column describes the type and size of the return arguments. An entry of n signifies a single integer value and an entry of n() implies an array of integer values. Values r and r() signify single and multiple real values, and l and l() imply single and multiple logical values. The logical values are represented by integers with zero meaning 'false' and any other value meaning 'true'. The global constant IDI__MAXCP can be used to dimension the arrays passed to the query routines; this is defined in the file 'IDI_PAR'.

Capability	Integer Code	Mnemonic	Output array
implementation level	1	IIMPLE	n
number of available configurations	10	INCON	n
selected configuration number	11	ICONS	n
physical size of display (x,y)	12	ISIZED	n(2)
display depth (number of bits in DAC)	13	IDISDE	n
maximum depth of LUTs	14	ILUTDE	n
number of LUTs	15	INLUT	n
number of ITTs per image memory	16	INITT	n
zoom range (min,max)	17	IZOOMR	n(2)
number of available VLUT colours	18	INLUTC	n
memory visible	20	IMEMVI	l()
list of memories in currently selected configuration	21	IMEMS	n()
depth of memories (in bits)	22	IMEMDE	n()
list of current LUT bindings	23	ILUTBI	n()
list of current display path bindings	24	IDISBI	n()
list of current ITT bindings	25	IITTBI	n()
maximum dimension of transfer window in x	30	ITWMDX	n()
maximum dimension of transfer window in y	31	ITWMDY	n()
transfer window x sizes	32	ITWSIX	n()
transfer window y sizes	33	ITWSIY	n()
transfer window x offsets	34	ITWOFX	n()
transfer window y offsets	35	ITWOFY	n()
depth of transfer windows (in bits)	36	ITWDE	n()

Capability	Integer Code	Mnemonic	Output array
number of available device cursors	40	INCUR	n
array of cursor shapes	41	IACUSH	n()
number of cursor shapes	42	INCUSH	n
list of current cursor bindings	43	ICURBI	n()
list of current cursor shapes	44	ICURSH	n()
list of current cursor colours	45	ICURCO	n()
list of current cursor visibilities	46	ICURVI	l()
number of locators	50	INLOC	n
number of real evaluators	51	INREVL	n
number of integer evaluators	52	INIEVL	n
number of logical evaluators (switches)	53	INLEVL	n
number of character evaluators	54	INCEVL	n
number of triggers	55	INTRIG	n
ROI implemented	60	ISROI	l
number of device ROIs	61	INROI	n
list of current ROI bindings	62	IROIBI	n()
list of current ROI marker colours	63	IROICO	n()
list of current ROI visibilities	64	IROIVI	l()
blink implemented	70	ISBLI	l
blink period for each memory	71	RBLIP	r()
trigger number to increase blink rate	77	IBLIIT	n
trigger number to decrease blink rate	78	IBLIDT	n
trigger number to stop blink	79	IBLIST	n
split screen implemented	80	ISSS	l
split screen x memory offsets	81	ISSOFX	n()
split screen y memory offsets	82	ISSOFY	n()
split screen (x,y) address	83	ISSXY	n(2)
split screen enabled	84	ISSSON	l

B ADAM Programmer's Guide to the IDI package

B.1 Introduction

This section describes the use of the Image Display Interface (IDI) in ADAM application programs. It is expected that the reader is familiar with programming for ADAM and with IDI.

The image display interface (IDI) is fully described in the specification document, and the details of the Starlink implementation are given in the preceding sections. This section will therefore only deal with issues relating to the use of IDI in ADAM application programs.

All IDI routines may be used in ADAM applications with the exception of the following routines, which must **never** be called in ADAM programs

IIDOPN - **Open Display**
IIDCLO - **Close Display**

The function of IIDOPN is performed by the environment routine IDI_ASSOC. The function of IIDCLO is performed by the environment routine IDI_CANCL, or IDI_ANNUL.

Note that these routines, unlike the core IDI routines, use the principle of inherited status. The routine IDI_ASSOC will only execute if the status is equal to zero on entry. The routines IDI_ANNUL and IDI_CANCL will execute if the status is non-zero on entry, so that devices can be closed in the event of unrelated errors, and will return the error status unchanged.

B.2 The IDI Parameter Routines

As with all other packages in the Software Environment, the only access to objects outside of application programs is via Program Parameters. IDI has three subroutines (the IDI parameter routines) that provide the necessary interaction with the outside world. They are:

IDI_ANNUL(dispid, status)
    Associate an IDI device with a parameter

IDI_ASSOC(pname, acmode, dispid, status)
    Release an IDI device

IDI_CANCL(pname, status)
    Release an IDI device and cancel the parameter

Here is a skeletal example of a program using IDI

```
      SUBROUTINE IDI_TEST ( STATUS )

      INCLUDE 'IDI_ERR'
      INTEGER ID, STATUS

*   Open a display device for IDI and reset it
      CALL IDI_ASSOC( 'DEVICE', 'WRITE', ID, STATUS )
      IF ( STATUS .EQ. IDI__OK ) THEN
```

```
*   Perform IDI functions on this display
         ...

*   Release device
         CALL IDI_ANNUL( ID, STATUS )
       ENDIF
```

IDI_ASSOC should be the first IDI routine to be called in the application. It obtains the name of the graphics device to be used (via the parameter system) and opens this device for subsequent IDI operations.

The first argument of IDI_ASSOC is a Program Parameter (which should be defined to be a device parameter in the Interface Module for the application).

The second argument is the access mode required. This can be one of

'READ' The application is only going to 'read' from the device, (i.e. perform cursor or similar operations), without altering the display appearance. The display is not cleared or reset when the device is opened.

'WRITE' The application is going to 'write' on the device creating a new screen display. The display is cleared and reset when it is opened.

'UPDATE' The application will modify the device, taking the existing display and changing it. The display is not cleared or reset when the device is opened.

The third argument is the display identifier returned to the application. This must be passed to all subsequent IDI routines which are to be used on this device.

The fourth argument is the usual status value.

When the application has finished using the device, the display is closed and the identifier annulled using the IDI_ANNUL, or IDI_CANCL routines.

B.3 Interface File

A simple interface file for the above example would be:

```
interface IDI_TEST
   parameter DEVICE
      position 1
      ptype  'DEVICE'
      keyword 'DEVICE'
      access 'READ'
      vpath 'PROMPT'
      default IKON
      prompt 'Display device '
   endparameter
endinterface
```

B.4 Reference Section

The description of each of the application level subroutines in the package now follows

IDI_ANNUL
Annul IDI display identifier and close the device.

The device associated with the display identifier is closed and the identifier is annulled. No further IDI routines can be called with this display identifier.

CALL IDI_ANNUL (DISPID, STATUS)
DISPID = INTEGER (given)
A variable containing the display identifier.
STATUS = INTEGER
Variable to contain the status. This routine is executed regardless of the given value of status. If its input value is not IDI__OK then it is left unchanged by this routine, even if it fails to complete. It its input value is IDI__OK and this routine fails, then the value is changed to an appropriate error number.

IDI_ASSOC
Open a graphics device and return a display identifier.

Associate a graphics device with the specified parameter and return a display identifier to reference this device.

CALL IDI_ASSOC (PNAME, ACMODE, DISPID, STATUS)
PNAME = CHARACTER * (*) (given)
String specifying the name of the device parameter.
ACMODE = CHARACTER * (*) (given)
String specifying the access mode: 'READ', 'WRITE' or 'UPDATE' as appropriate.
DISPID = INTEGER (returned)
A variable containing the display identifier.
STATUS = INTEGER
Variable to contain the status. If this variable is not IDI__OK on input, then the routine will return without action. If this routine fails to complete, this variable will be set to an appropriate error number.

IDI_CANCL
Close an IDI display and break the parameter association.

The device associated with the parameter is closed and the association between the graphics device and the specified device parameter is broken.

CALL IDI_CANCL (PNAME, STATUS)
PNAME = CHARACTER * (*) (given)
String specifying the name of the device parameter which has previously been associated with a device using IDI_ASSOC.
STATUS = INTEGER
Variable to contain the status. This routine is executed regardless of the given value of status. If its input value is not IDI__OK then it is left unchanged by this routine, even if it fails to complete. If its input value is IDI__OK and this routine fails, then the value is changed to an appropriate error number.

C Example Program

This example program will read in a data file and display it in the middle of the screen. A cursor will be displayed which can be moved about the screen with the mouse, and the image can be zoomed and unzoomed using the left hand and centre mouse buttons. The interaction is terminated by pressing the right hand mouse button, and the cursor position relative to the memory origin is output.

The input data file should contain integers between 0 and 255.

```
      SUBROUTINE IDI_TEST ( STATUS )
*+
*  Name:
*     IDI_TEST
*  Purpose:
*     TEST IDI
*  Language:
*     FORTRAN
*  Type of Module:
*     ADAM A-task
*  Arguments:
*     STATUS = INTEGER (Given and Returned)
*        The global status.
*  Authors:
*     NE: Nick Eaton  (Durham University)
*  History:
*     May 1989 (NE):
*        Original version
*     March 1991 (NE):
*        Update to use NDFs and work with X-windows interface
*-
*  Type definitions:
      IMPLICIT NONE             ! No implicit typing
*  Global Constants:
      INCLUDE 'IDI_ERR'         ! IDI error codes
      INCLUDE 'IDI_PAR'         ! IDI global constants
*  Status:
      INTEGER STATUS            ! Global status
*  Local variables:
      CHARACTER * 64 TEXT
      INTEGER TRIGS( IDI__MAXTR )
      INTEGER DEPTH, DIRECN, DSIZE(2), EXTRN, ID, INDF, INTID, INTOP,
     :        INTTY, IPIN, LBND(2), LTEXT, MEMID, NARR, NCHAR, NDATA,
     :        NDIM, NEL, NUMCUR, NVAL, NX, NY, OBJID, OBJTY, OUTMID,
     :        PACK, UBND(2), XC, XOFF, XSTART, YC, YOFF, YSTART
*.

*   Check inherited global status.
      IF ( STATUS .NE. IDI__OK ) GOTO 99

*   Get the picture to plot. Should be 2-dimensional.
      CALL NDF_BEGIN
      CALL NDF_ASSOC( 'IN', 'READ', INDF, STATUS )
      CALL NDF_MAP( INDF, 'DATA', '_INTEGER', 'READ', IPIN, NEL, STATUS)
      CALL NDF_BOUND( INDF, 2, LBND, UBND, NDIM, STATUS )
      IF ( ( STATUS .NE. IDI__OK ) .OR. ( NDIM .GT. 2 ) ) GOTO 99

*   Calculate image size
      NX = UBND( 1 ) - LBND( 1 ) + 1
      NY = UBND( 2 ) - LBND( 2 ) + 1
      NDATA = NX * NY

*   Open IDI for a device obtained through the parameter system
*   Force a reset by using 'WRITE' mode
      CALL IDI_ASSOC( 'DEVICE', 'WRITE', ID, STATUS )
      IF ( STATUS .NE. IDI__OK ) GOTO 99

*   Inquire the physical size of the display using Query Capabilities
*   Integer code 12 ( ISIZED ) returns the screen size in pixels
      NARR = 2
      CALL IIDQCI( ID, ISIZED, NARR, DSIZE, NVAL, STATUS )

*   Check the image size against the display size
      IF ( ( NX .GT. DSIZE( 1 ) ) .OR. ( NY .GT. DSIZE( 2 ) ) ) GOTO 99

*   Inquire the depth of the first (default) memory using Query Capabilities
*   Integer code 22 ( IMEMDE ) returns the memory depth
      NARR = 1
      CALL IIDQCI( ID, IMEMDE, NARR, DEPTH, NVAL, STATUS )

*   Set up the transfer window to be the same size as the input image
*   and to be centered in the middle of the screen; XOFF, YOFF define
*   the position of the lower left corner of the transfer window. Plot
*   the image in the base memory ( 0 ) and loading from bottom to top ( 0 )
      MEMID = 0
      DIRECN = 0
      XOFF = MAX( 0, ( DSIZE( 1 ) - NX ) / 2 )
      YOFF = MAX( 0, ( DSIZE( 2 ) - NY ) / 2 )
      CALL IIMSTW( ID, MEMID, DIRECN, NX, NY, DEPTH, XOFF, YOFF,
     :             STATUS )

*   Write the image to the screen starting at position 0, 0 in the
*   transfer window. The data is taken from the lowest byte of each
*   integer word in the data array ( defined by PACK = 1, DEPTH = 8 ).
      PACK = 1
      XSTART = 0
      YSTART = 0
      CALL IIMWMY( ID, MEMID, %VAL( IPIN ), NDATA, DEPTH, PACK,
     :             XSTART, YSTART, STATUS )

*   Abort if an error has occurred
      IF ( STATUS .NE. IDI__OK ) THEN

*   Obtain a meaningful IDI error message
         CALL IIDERR( STATUS, TEXT, NCHAR )
```

```
*   Output this message
         CALL ERR_REP( 'IDI_WMY', TEXT, STATUS )
         GOTO 99
      ENDIF

*   Display the memory by setting its visibility to .TRUE.
      CALL IIMSMV( ID, MEMID, 1, 1, STATUS )

*   Set up interactions to move the cursor and zoom the memory
*   Set up the mouse ( interactor type = 0, interactor id = 0 ) to
*   control the cursor ( object type = 1, object id = 0 ) by moving
*   it ( interactive operation = 1 ). End the interaction by pressing
*   the right hand button ( exit trigger number = 2 ).
      INTTY = 0
      INTID = 0
      OBJTY = 1
      OBJID = 0
      INTOP = 1
      EXTRN = 2
      CALL IIIENI( ID, INTTY, INTID, OBJTY, OBJID, INTOP, EXTRN,
     :             STATUS )

*   Inquire the interactive operation and instruct the user
      CALL IIIQID( ID, INTTY, INTID, TEXT, LTEXT, STATUS )
      TEXT( LTEXT + 1 : ) = ' to control cursor'
      CALL MSG_OUT( ' ', TEXT, STATUS )

*   Set up the left hand button ( interactor type = 5, interactor id = 0 )
*   to control the memory ( object type = 5, object id = 0 ) by increasing
*   the zoom ( interactive operation = 3 ). End the interaction by pressing
*   the right hand button ( exit trigger number = 2 ).
      INTTY = 5
      INTID = 0
      OBJTY = 5
      OBJID = MEMID
      INTOP = 3
      EXTRN = 2
      CALL IIIENI( ID, INTTY, INTID, OBJTY, OBJID, INTOP, EXTRN,
     :             STATUS )

*   Inquire the interactive operation and instruct the user
      CALL IIIQID( ID, INTTY, INTID, TEXT, LTEXT, STATUS )
      TEXT( LTEXT + 1 : ) = ' to increase zoom'
      CALL MSG_OUT( ' ', TEXT, STATUS )

*   Set up the centre button ( interactor type = 5, interactor id = 1 )
*   to control the memory ( object type = 5, object id = 0 ) by decreasing
*   the zoom ( interactive operation = 4 ). End the interaction by pressing
*   the right hand button ( exit trigger number = 2 ).
      INTTY = 5
      INTID = 1
      OBJTY = 5
      OBJID = MEMID
      INTOP = 4
      EXTRN = 2
      CALL IIIENI( ID, INTTY, INTID, OBJTY, OBJID, INTOP, EXTRN,
     :             STATUS )

*   Inquire the interactive operation and instruct the user
      CALL IIIQID( ID, INTTY, INTID, TEXT, LTEXT, STATUS )
      TEXT( LTEXT + 1 : ) = ' to decrease zoom'
      CALL MSG_OUT( ' ', TEXT, STATUS )

*   Inquire the exit trigger operation and instruct the user
      CALL IIIQID( ID, INTTY, EXTRN, TEXT, LTEXT, STATUS )
      TEXT( LTEXT + 1 : ) = ' to exit'
      CALL MSG_OUT( ' ', TEXT, STATUS )

*   Move the cursor to the middle of the screen. A memory id = -1 sets
*   the cursor position relative to the screen origin.
      XC = DSIZE( 1 ) / 2
      YC = DSIZE( 2 ) / 2
      NUMCUR = 0
      CALL IICINC( ID, -1, NUMCUR, 1, 2, XC, YC, STATUS )

*   Display the cursor by setting its visibility to .TRUE.
      CALL IICSCV( ID, NUMCUR, 1, STATUS )

*   Execute the interactions.
      CALL IIIEIW( ID, TRIGS, STATUS )

*   Read the cursor position relative to the memory origin.
      CALL IICRCP( ID, MEMID, NUMCUR, XC, YC, OUTMID, STATUS )

*   Output the cursor position.
      CALL MSG_SETI( 'XC', XC )
      CALL MSG_SETI( 'YC', YC )
      CALL MSG_OUT( ' ', 'Cursor position is ^XC , ^YC', STATUS )

*   Undisplay the cursor by setting its visibility to .FALSE.
      CALL IICSCV( ID, NUMCUR, 0, STATUS )

  99  CONTINUE

*   Close down IDI using the parameter system
      CALL IDI_CANCL( 'DEVICE', STATUS )

*   Close the data file
      CALL NDF_END( STATUS )

      END
```

```
interface IDI_TEST
   parameter IN
      position 1
      type 'NDF'
      keyword 'IN'
      access 'READ'
      vpath 'PROMPT'
      prompt 'NDF file containing image '
   endparameter
   parameter DEVICE
      position 2
      ptype  'DEVICE'
      keyword 'DEVICE'
      access 'READ'
      vpath 'PROMPT'
      default XWINDOWS
      prompt 'Display device '
   endparameter
endinterface
```

D X-WINDOWS implementation notes

D.1 GWM interface

IDI uses the Graphics Window Manager to look after the creation and management of the display window. When IDI opens (IIDOPN or IDI_ASSOC) it first looks for an existing GWM window with the given name, and if it finds one it attaches to that, otherwise it creates a new window with the default characteristics. The window pixmap, which is used by GWM to store the picture, is then read by IDI and stored in an internal memory. The purpose of this is to enable existing displays to be manipulated by IDI, for example a picture drawn with GKS can be scrolled using IDI. In an ADAM application 'READ' or 'UPDATE' mode should be used in IDI_ASSOC if an existing display is to be manipulated.

If an application wants to start with a clean sheet then the display should be reset. In an ADAM application this is done by specifying 'WRITE' mode in IDI_ASSOC. In a non-ADAM application the routine **Reset Display** should be called immediately after **Open Display**.

D.2 Memories

When an X-window device is opened one memory is made available (identifier 0) which corresponds to the window pixmap. The size of the memory is equal to the size of the pixmap, which can be different to the size of the window.

D.3 Overlays

If a GWM window contains an overlay plane then this can be accessed as if it were a separate memory (identifier 1). The overlay memory depth is only one bit but it can be accessed like any other image memory. It can be written to with **Write Memory**, **Polyline** or **Plot Text**. It can be scrolled and zoomed separately to the base memory. It has a separate look-up table (identifier 1) to the base plane. Although the memory is only one bit deep the overlay LUT has as many entries as the base LUT, however every entry contains the same overlay colour. If the overlay colour is to be changed then the new colour should be sent to all the pens in the overlay LUT.

D.4 Writing to the memories

Most workstations running IDI will have an 8 bit display, but the full 8 bit capability of the device is not normally available. The X-windows driver will reserve a number of pens for its exclusive use and GWM will not normally allocate all the available pens to one window. The look-up table length is adjusted by IDI to match the number of pens allocated to the GWM window. The default intensity transformation table is adjusted, so that data values going from 0 to 255, map linearly onto the look-up table going from 0 to the allocated length. For instance, if the look-up table contains 64 entries then a data value of 0 will be assigned to pen 0 and a data value of 255 will be assigned to pen 63, with intermediate values scaled appropriately. The number of pens allocated to the window can be inquired using **Query Capabilities Integer** with capability number 18 (INLUTC), or can be inquired from GWM using GWM_GETCI.

The input to the image display routines **Write Memory**, **Read Memory** is an array of integers, and the arguments *packing factor* and *data depth* control how the data to be displayed is packed into a single integer word.

D.5 Zoom

The X driver does not support hardware zooming and so the zooming has been implemented in software. The current implementation allows zooming by factors of up to 32 times. The zoom factors go from -31 to 31, with 0 as no zoom. Negative zooming (zoom factors less than 0) is not currently supported for *interactive operations*.

D.6 Look-up tables

Each window has one look up table (LUT) and one intensity transformation table (ITT) per memory. The intensity transformation table is used to translate pixel numbers into entries in the look up table.

D.7 Interactions

The X-windows interface supports two *interactor types*; *locators* and *triggers*. Two types of *locator* are supported: the mouse *interactor identifier* = 0 and the keyboard arrows *interactor identifier* = 1. The action of the keyboard arrows can be speeded up by simultaneously depressing the SHIFT key. When using the keyboard as an interactor the window focus has to be shifted to the display window by clicking in the window or on the title bar.

The number of available triggers can be inquired using capability code 55 (INTRIG) in **Query Capabilities Integer**. Triggers can be either the mouse buttons (identifiers 0, 1, 2), or keys on the keyboard (identifiers 3 to 54). The table lists the current set of triggers and the corresponding identifiers.

Id	Trigger	Id	Trigger	Id	Trigger
0	Mouse button left	20	G key	40	0 key
1	Mouse button centre	21	H key	41	1 key
2	Mouse button right	22	I key	42	2 key
3	Up arrow	23	J key	43	3 key
4	Up arrow shifted	24	K key	44	4 key
5	Left arrow	25	L key	45	5 key
6	Left arrow shifted	26	M key	46	6 key
7	Right arrow	27	N key	47	7 key
8	Right arrow shifted	28	O key	48	8 key
9	Down arrow	29	P key	49	9 key
10	Down arrow shifted	30	Q key	50	Period key
11	Keypad PF4	31	R key	51	Plus key
12	Space bar	32	S key	52	Minus key
13	Return key	33	T key	53	Shift key left
14	A key	34	U key	54	Shift key right
15	B key	35	V key		
16	C key	36	W key		
17	D key	37	X key		
18	E key	38	Y key		
19	F key	39	Z key		

A user can be informed of the appropriate trigger by obtaining a description from the routine **Query Interactor Description**.

At present only a few of all the possible interactions have been implemented. A non-zero error status will be returned from **Execute Interaction and Wait** if the interaction has not been implemented. The table shows the implemented interactions with a cross (+).

X-windows Interaction Cross Reference							
Description	No effect	Cursor	ITT	VLUT	ROI	Memory	Display
Application specific code	+	.	.	.	.	.	.
Move object	+	+	.	.	+	+	+
Rotate object	+	.	.	+	.	.	.
Increase zoom	+	.	.	.	.	+	+
Reduce zoom	+	.	.	.	.	+	+
Set zoom to normal	+	.	.	.	.	+	+
Blink object	+	.	.	.	.	.	.
Modify object	+	.	.	.	+	.	.

Scrolling a memory or the display with the mouse begins as soon as the hardware pointer enters the display window, and the memory moves as if it were attached to the hardware pointer, however scrolling does not happen if the mouse is moved too quickly.

With the X-windows interface it is possible to display a fixed cursor and to scroll and zoom the memory under it, although the default hardware cursor should not be used in this case. A fixed crosshair cursor or region of interest is made visible at the required position, and an interaction to scroll the memory is set up. The example program can easily be converted to perform this operation. To do this replace the cursor scroll interaction with a memory scroll interaction. Thus

```
      INTTY = 0
      INTID = 0
      OBJTY = 1
      OBJID = 0
      INTOP = 1
      EXTRN = 2
      CALL IIIENI( ID, INTTY, INTID, OBJTY, OBJID, INTOP, EXTRN,
     :             STATUS )
```

becomes

```
      INTTY = 0
      INTID = 0
      OBJTY = 5
      OBJID = MEMID
      INTOP = 1
      EXTRN = 2
      CALL IIIENI( ID, INTTY, INTID, OBJTY, OBJID, INTOP, EXTRN,
     :             STATUS )
```

When rotating the look-up table the action is controlled by the left and right movement of the mouse, or with the left and right keyboard arrows.

D.8 Cursors

The default cursor (*cursor shape* = 0) is the hardware pointer. When it is activated, and in the display window, it is shown by an open cross.

The cross hair cursor (*cursor shape* = 1), cross cursor (*cursor shape* = 2) and regions of interest (ROI) are drawn onto the screen using an exclusive OR (XOR). When using the mouse to control the movement of one of these cursors the hardware pointer has to be brought close to the centre of the crosshairs. When close enough the crosshair cursor becomes attached to the hardware pointer and will remain attached as long as the mouse is moved slowly. If the mouse is jerked quickly the hardware pointer and crosshair will part. When using a region of interest the hardware pointer has to be brought close to the active corner.

D.9 Text

Text is drawn with one of the preset device fonts. The text sizes small, normal and large are distinct, but the very large text size is the same as the large size. Only the default text path and text orientation are supported.

D.10 Deficiencies

- Cursors or regions of interest drawn with the XOR function may sometimes be hard to see if many of the allocated pens used to draw an image contain similar colours. Installing a more varied look-up table should make the cursors more visible.
- When scrolling a memory, text or polylines drawn by the current application will not be refreshed if that part of the memory is scrolled out of the window and then scrolled back, unless the window is completely refreshed by, for example, zooming.