Putting a new face on APL2

by J. R. Jensen
K. A. Beaty

APL2/X is an interface between APL2 and the X Window System®, built at the IBM Cambridge Scientific Center. This interface enables the full set of the X Window System Xlib calls and the related data structures to be used directly from programs written in APL2, thereby providing APL2 with a true, full-function windowing environment. The interface also deals with the broader and more general issue of how to call C programs from APL2. The interface and the experience of building it are described in some detail in this paper.

The intent of this paper is to detail the experience of building an interface between APL2 and the X Window System**. APL2, having evolved over two and a half decades, was a good candidate for a "face lift" in that it benefits greatly from having a modern presentation system. In turn, the X Window System gains the flexibility and power of APL2 in developing and driving applications.

This paper is divided into several subsections. To set the stage, some simple examples of how the interface can be used are shown, and an overview of the X Window System is also given. With that as a background, we then discuss the rationale for building such an interface, as well as some of the design choices made. Next, the general APL2-to-C interface that has been implemented is presented. APL2/X uses this interface heavily. The focus is on how to be able to use a large number of already-existing C routines from APL2 with as little additional work required as possible. Finally, examples of how to use this interface to access and call C routines from APL2 are shown, with a focus on the special consideration that the X Window System entails.

It is assumed that the reader has some knowledge of both APL2 and the X Window System. See, for example, *APL2 at a Glance* by Brown et al.[1] for an introduction to APL2, and *Introduction to the X Window System* by Jones[2] for information about the X Window System.

An example. An example might help illustrate the capabilities of the X Window System when used with APL2. To display the image of this example, run the following APL2 expression:

```
XIMAGE MAN COL 'Basic'
```

XIMAGE is an APL2 function that uses the X Window System calls to display an image. *MAN* is an APL2 variable containing an image of a mandrill, *COL* is a color lookup table, and *'Basic'* is a window title. It results in a new window displaying the content shown in Figure 1.

Figure 1 Basic image

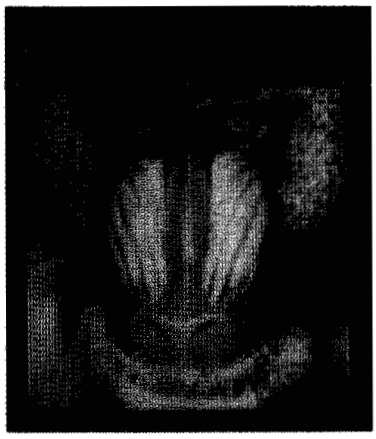
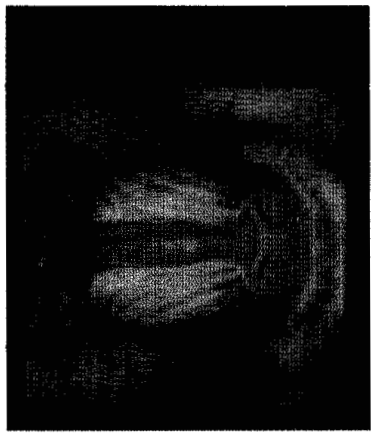


Figure 2 Image turned on side



This image can be manipulated using normal APL2 functions. The manipulation can take place on the image matrix, the color table, or both. For instance, to turn it on its side as in Figure 2 use:

```
XIMAGE (⍉MAN) COL 'Lazy'
```

To triple its size as shown in Figure 3, use the following function:

```
XIMAGE (3/3⌿MAN) COL 'Large'
```

To display as a negative as is done in Figure 4 use:

```
XIMAGE MAN (1000-COL) 'Neg'
```

Finally, to create four mirror images of the mandrill as in Figure 5 use:

```
B2←MAN,⌽MAN
B4←B2,[1] ⊖B2
XIMAGE B4 COL 'Four'
```

Why an APL2 X Window System interface. From the advantages each has to offer, it is evident that APL2 and the X Window System can benefit from an interface connecting them. We now describe some of the more compelling benefits for APL2.

APL2 is provided with a modern-day interface. The present interface of APL2 dates back to the late 1970s and has a distinct character-cell flavor to it. Graphics are limited to fixed, nonmovable images. Several desirable features can be incorporated by utilizing the functions of the X Window System:

- Keystroke sensitivity for programs
- Pointing devices such as a mouse
- Multiple fonts of varying size
- Bitmapped graphics and image
- Dynamic graphics capabilities

Many of these features are as much a product of better hardware (in the form of workstations) as they are of the software, but this does not negate the fact that they need the software to utilize these advanced features.

The interface enables a given APL2 application to display its output on any connected workstation, and enables a workstation to initiate and run APL2 programs on many different hosts at the same time.

Similarly, the X Window System gains from using APL2. APL2 provides an interactive environment. Each call or series of calls can be tried out, verified, and altered at will until the right combination is reached. This activity can take place without any recompilation whatsoever, speeding up the development process. The X Window System can use the array processing ability of APL2 to easily store and manipulate images using standard APL2 primitives, as shown in the example presented earlier.

For those readers not familiar with the X Window System, the next section presents a brief overview.

An overview of the X Window System

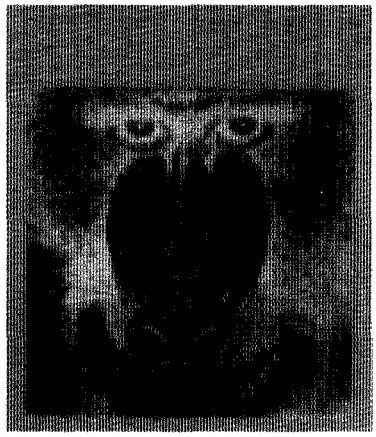
The X Window System is the *de facto* standard for windowing systems in the UNIX** environment. In

Figure 3 Image tripled in size

many respects it is very similar to the Operating System/2* (OS/2*) Presentation Manager* and Microsoft Windows** for the IBM Personal Computer Disk Operating System (PC DOS) in that it provides the application programmer with a multitude of calls to control and manipulate the content of windows on a display. However, it also differs from these products in some key aspects. The foremost difference is that the X Window System was designed from its inception to be network-transparent. This means that an application can display its results on any workstation attached to a local area network, no matter where the application may actually be running. The X Window System employs the client-server model of computing. It enables the application, or client, to make use of the resources of the workstation, or server, to display its output and receive input from the user, as illustrated in Figure 6.

Server. The X Server is a program running on the workstation that manages the interaction with the user. It typically controls one or more screens, a keyboard, and a mouse or similar pointing device. It allows clients to have use of all of these devices and other resources such as windows, pixmaps, fonts, and graphics contexts. The server receives directives from communicating clients via network

Figure 4 Negative of image



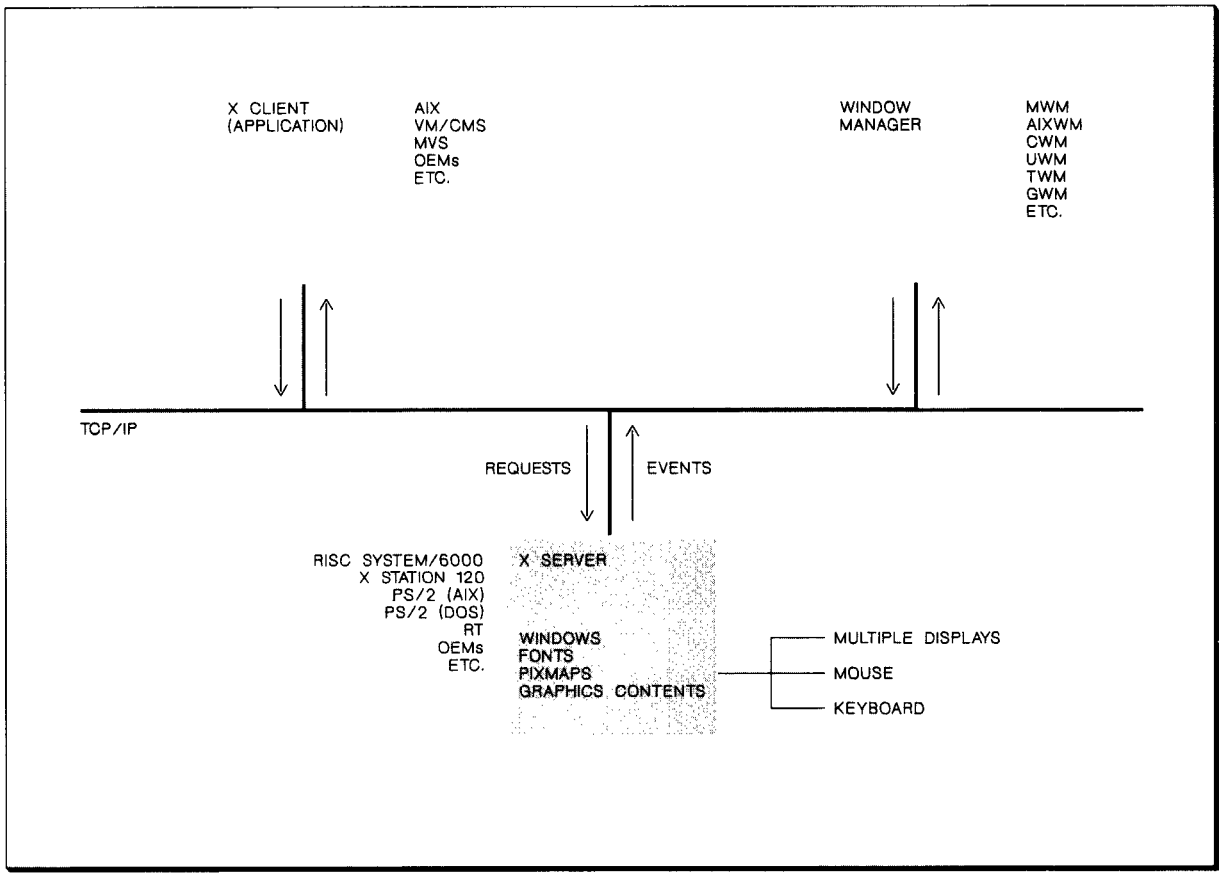
protocol requests and acts upon them to draw windows, graphics, text, and images on the display. Whenever the user of the workstation performs an action such as pressing a key, moving the mouse, etc., the server will generate an event message and return it to the client program via the underlying network protocol (traditionally TCP/IP, the Transmission Control Protocol/Internet Protocol).[3]

Client. The application is the client program. It sends requests to the server via the network protocol and receives information back from the server in the form of replies or events. These requests can be generated and handled at the network protocol level, or higher-level calls can be used.

Window manager. The window manager is an application that controls where windows are placed, the size of the windows, the window decorations, and the interaction style. In separating it from the X server, the X Window System has made it possible to have a replaceable window manager implementing different interaction styles. As an example, some window managers enable window moving and resizing by grabbing and dragging the window borders, whereas other window managers will use menus to accomplish the same end result. Among the many window managers that exist, one now in common use is the Open Software Foundation (OSF) Motif** window manager that gives the X Window System a "look and feel" almost

Figure 5 Image quadrupled

Figure 6 The X Window System client-server relationship



identical to that of the OS/2 Presentation Manager. A list of other existing window managers can be found in Figure 6.

These three components of the X Window System need not run on the same processor. An application can be running on, say, a host with the Virtual Machine/Extended Architecture operating system, or VM, communicating through the X Window System client services with an X Window System server running on an IBM RISC System/6000*. The net effect of this setup is that the results of the application appear on the display of the workstation as though the application had been run locally.

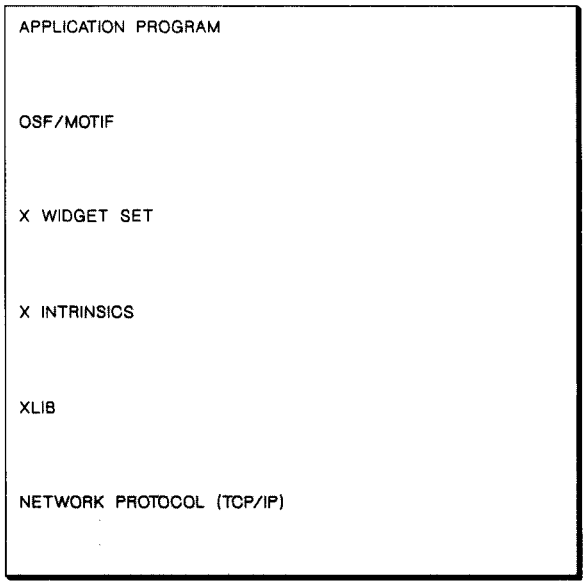
It is also possible for a single client application to display on many servers at once. Likewise, a server can service many clients at the same time, displaying the output of each application program at once.

Communications among the client, server, and window manager can be handled by any method that provides a reliable bidirectional byte stream. When the client and server both run on the same processor, some sort of interprocess communication is used for communication between the two. When the client and server are running on separate processors, the TCP/IP communications protocol usually provides this service, although any other reliable communications scheme could potentially be used in its place.

The X Window System output capabilities can be summarized as follows:

- Controlling multiple windows on one or more display screens
- Drawing graphics primitives such as lines, arcs, rectangles, and polygons with or without fills

Figure 7 The X Window System hierarchy



- Writing high-quality text with many different fonts
- Supporting images

Inputs are received in the form of events. Events can be generated by the user pressing a key on the workstation keyboard, by the user manipulating the mouse (moving it or using the mouse buttons), or by other events, such as when a window is cleared and needs to be redrawn.

The X Window System is a layered architecture, depicted in Figure 7. An application can draw upon the calls of all of these layers. The X Window System *network protocol* is at the base of the hierarchy, ultimately defining the traffic flowing between the client and server components.

The *Xlib* level is the next level. Most application programmers will never interface with the X Window System at a level lower than this one. It consists of about 400 separate calls written in C and more than 100 data structures.

The *X Intrinsics* and the *X Widget Set* taken together form the *X Toolkit*. A widget set is a collection of common graphics elements that applications may use, such as menus, scrollbars, pop-up windows, and the like. The X Widget Set makes use of the X Intrinsics, which provides it with an object-oriented interface.

The OSF/Motif toolkit is an elaborate toolkit that implements application elements such as sliders, pull-down menus, and buttons in a three-dimensional appearance.

One final aspect of the X Window System needs to be touched upon. It does not seem to be a generally known fact that it is indeed possible to run the X Window System under VM[4] or Multiple Virtual Storage (MVS).[5] In fact, most of the development work of APL2/X was performed on a VM system. VM and MVS both support the X client services as part of TCP/IP Version 2 for VM and TCP/IP Version 2 for MVS.

Interface design criteria

As a stepping-stone to building the X Window System interface, a general APL2-to-C interface was implemented. Although general in scope, it is certainly true that its built-in functionality has been heavily influenced by the following considerations that surfaced during the construction of the APL2/X interface.

- The interface should be able to use the existing C functions without any changes or modifications. This requirement is important, since the source code for the C functions may not be available.
- The interface must be able to support a large number of calls efficiently. The X Window System defines about 400 separate calls, depending on the release considered.
- To be useful, APL2/X must be able to support data structures but also allow APL2 to manipulate the data using APL2 functions. Data structures play an important role in many X Window System calls.
- The external C routines must be used in a manner much akin to normal APL2 functions (i.e., maintain the "feel" of an APL2 function) when called from within APL2. Specifically, attainment of this likeness requires that function arguments be passed explicitly and by value. The interface must take care of the needed argument type coercion. Also, the interface should specifically re-

frain from using designated variables or storage areas that can be updated as a side effect of the call; rather, all output should be returned as explicit results of the call.

- The defined X Window System call syntax should be adhered to as closely as possible so as to enable the use of the normal X Window System documentation. Only two noteworthy deviations apply throughout the interface:

  1. Output-only arguments are never specified on input; they will be generated automatically and returned as part of the explicit result of the function call.
  2. Arguments are given by value, even in cases such as a character string, where C expects a pointer in the parameter list. The interface again handles the details of making this happen.

  At times, maintaining this fidelity to the X Window System call syntax seems slightly out of place in an APL2 setting. One of the places where this is apparent is on those calls where the X Window System expects a varying number of arguments passed in an array or character string. These calls invariably require the specification not only of the array itself, but also of the number of elements in the array. This latter piece of information is, of course, directly available with the APL2 array, so it seems slightly silly and annoying to have to specify it in the call. However, in the name of consistency we have chosen to stay with the X Window System call syntax throughout, even in cases such as this one.

- APL2 will handle storage management automatically, whereas C most often leaves the task for the caller to do. APL2/X takes over this chore when calling the C functions, so the APL2 program is freed from addressing this task explicitly.

- Enable the same interface from APL2 to the X Window System in multiple host environments to allow APL2 applications that use APL2/X to be run under the Virtual Machine/Conversational Monitor System (VM/CMS), Multiple Virtual Storage/Time-Sharing Option (MVS/TSO), or under Advanced Interactive Executive* (AIX*) on the RISC System/6000.

All of these items are discussed later in more detail.

Using the X Window System from APL2

APL2 can use the X Window System in two different ways. It can either use it indirectly, if the output device APL2 is communicating with is being remapped to a workstation running the X Window System, or directly by issuing calls to the X Window System from APL2. The indirect approach allows existing applications written for a 3270-type display screen to run on an X Window System workstation, but the interaction style is then, of course, limited to that of a 3270 device. However, to be able to utilize the features and facilities of the X Window System, it is necessary for the applications to be given direct, explicit access to the X Window System.

APL2 using the X Window System in compatibility mode. The simplest way today to use the X Window System from APL2 is in compatibility mode. Essentially it is another way of getting someone else to worry about supporting the X Window System. Two existing IBM products that do just that are described below.

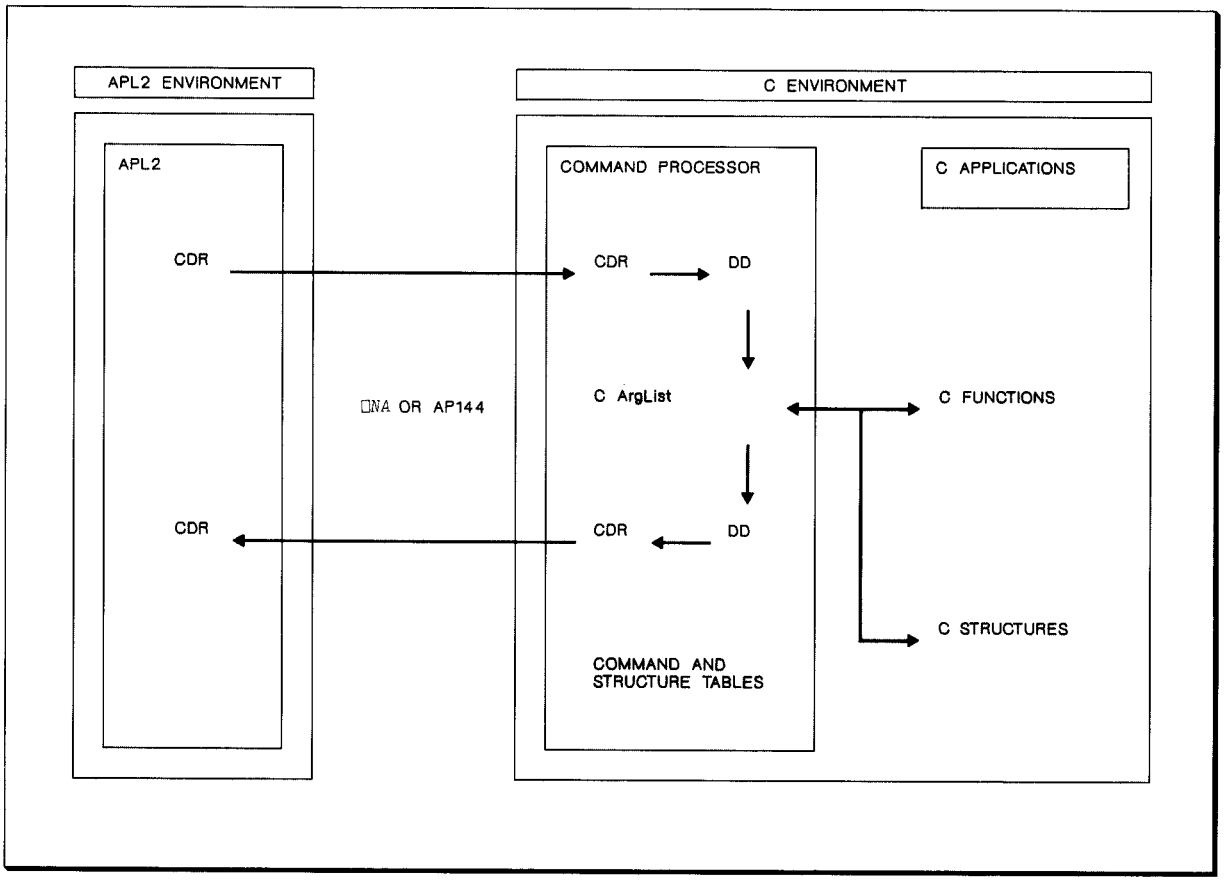
X3270. The X3270 is a terminal emulator that enables a 3270-type session to run on an X Window System workstation, given the proper network attachments. It supports different size fonts including APL2, GDDM-style graphics, and 3277GA emulation and has limited mouse support.[6]

GDDM/XD. GDDM/XD is an interface that permits the display of output from GDDM on workstations supporting the X Window System. It is available as part of TCP/IP Version 2 for VM and TCP/IP Version 2 for MVS. It displays both character and graphics output in a separate window on the X Window System workstation.[7]

Exploiting the X Window System from APL2. APL2/X takes a different approach to the X Window System. In order to fully exploit the X Window System from the APL2 environment, it is essential that the application be given direct access to all of the X Window System calls.

The connection between C and APL2 is illustrated in Figure 8. APL2/X receives data from APL2 in its common data representation (CDR) format. The CDR format is a documented data format for APL2 external data. It includes not only the data, but also descriptive information about data type, rank, and dimensions. The format varies, depending on the

Figure 8 Calling C programs from APL2



host operating environment. The data are sent from APL2 to APL2/X using the associated processor 11 (VM) or a new auxiliary processor AP144 (AIX).

Once in APL2/X, the incoming CDR is transformed into a DD, or data descriptor, which is the data representation used internally by APL2/X in all of the host environments within which it operates. This transformation essentially involves breaking up the CDR into self-contained arrays connected via pointers. This data representation can be used directly for new functions specifically written to use this data structure.

However, it is more common to use already-existing C functions. To do so, the data must be in the form that the functions can use. The second transformation is then involved to build the *C ArgList*. The argument list is for the C function that is to be called. The ArgList format is also employed when accessing and using C data structures.

The conversion process is controlled by a command definition that describes the arguments required by a given command. These definitions are stored in *command tables*. The first argument in any call identifies the command to be executed. The tables are searched to locate the matching command definition.

APL2/X data descriptor

APL2/370,[8] APL2/6000, and APL2/PC[9] all pass data to C in the form of a monolithic block of data. This block includes not only the data, but also information describing the data type, rank, and dimension. APL2/X breaks up the block of data into its component pieces, storing the descriptor and data infor-

mation in separately allocated areas for each nesting level of the data. This division ultimately cuts down on the amount of data copying needed, and has also enabled APL2/X to extend the descriptors with additional information. The descriptors currently hold the following pieces of information:

`rc`	Element return code
`flags`	Assorted control flags
`refs`	C indirection count
`data`	Pointer to data values (or the data value in the case of a scalar)
`alloc`	Number of elements allocated
`xrho`	Number of elements stored
`rtl`	Data type
`rank`	Rank
`dims`	Array dimensions (zero, one, or more)

APL2/X adds the `rc`, `alloc`, `flags`, and `refs` items for its own use. The remaining items are extracted from the data passed from APL2.

The ability to use separately allocated items has proved to be very useful when constructing elaborate return values. As an example, see the result of the `GetConst` command given later in the section on support for C constants. The result consists of a three-column table. The first column consists of a character string, the second column another character string, and the third column a value that can be either numeric or yet another character string. This table is built in a bottom-up fashion, with each element being appended in turn. With use of the separately allocated items, it just becomes a question of keeping track of a set of pointers, whereas a monolithic approach would require a preliminary pass to determine the size of the final table, before the actual building of it could get under way.

Defining and calling C functions from APL2

The supported X Window System calls are defined in a command table, along with their parameter and result type codes. The type codes are used to validate argument inputs and to gather resultant output for returning to APL2. The X Window System command table, as well as the related X Window System structure definitions table, are compiled into the command interface written in C. Therefore, the interface that resides between APL2 and the actual C functions (commands) being called is the one responsible for validating input and checking for and returning expected function results.

By having the command and structure definitions reside in the C command interface, we can funnel all Xlib calls through a common APL2 function rather than having an APL2 function for each X Window System function call. This significantly reduces the number of X-Window-System-related functions that need to be present in the application workspace.

The APL2/X interface ends up being identical in VM and AIX. Two simple APL2 functions *C* and *X* hide the fact that communication between APL2 and APL2/X is handled by processor 11 in VM and by shared variables in AIX, giving APL2/X a single common interface to APL2 in all host environments. These functions with calls and parameter are given in the following box.

```
(rc [results]) ← C command [parm] ...

[results] ← X command [parm] ...
```

The terms in the box are defined below.

command — The name of the X Window System call to be invoked, specified as an APL2 character vector.

[parm] — All but a few of the X Window System calls require additional input parameters to be specified. These parameters are given after the name of the call itself, in the same order as listed in the X Window System documentation.

[results] — The output from the call (if any) is returned in the form of an explicit result. This result includes the X Window System explicit result (if any), as well as any implicit results passed back via output parameters given on the call.

rc — The command return code. Note that this is only returned when using the *C* function. *C* and *X* only differ from one another in the way they deal with error conditions. *C* passes back an error code as part of the function return. It is then up to the calling program to check this code and take appropriate action on a nonzero return code. *X* supplies a default error-handler to check the return codes as they are returned from each call to APL2/X. *X* will suspend operation in the function by issuing a "`□ES 0 1`"

event, if an error is encountered. The programmer then has a chance to correct the problem.

Using these two functions as the base interface allows easy portability of applications from host environment to host environment, without having to change the calls of the application to the X Window System. We have been able to run identical sample APL2/X applications under VM/CMS and AIX on the RISC System/6000 without changing a single line of APL2 code.

The X Window System calls `XOpenDisplay` and `XDrawLines` can serve to illustrate the close correspondence between X Window System calls issued from C and from APL2.[10,11] In C, the calls might look like the following:

```
int points[4][2]=
    {{10,10},{100,10},{10,100},{10,10}};

dp = XOpenDisplay("");

XDrawLines(dp,win,gc,points,4,0)
```

The same calls can be issued from APL2 (via APL2/X) as follows:

```
points ← 10 10  100 10  10 100  10 10

dp ← X 'XOpenDisplay' ''

X 'XDrawlines' dp win gc points 4 0
```

The call to `XDrawLines` obviously assumes that the parameters `dp`, `win`, and `gc` have been set up by preceding calls to other X Window System functions.

Command definitions. The interface can support an unlimited number of C routines. Each routine is defined by a command definition that describes the needed aspects of the call as follows:

- Command name
- Input type codes
- Output type codes
- Address of C function to be called
- Call method
- Two optional parameters that can be used by the command

Some examples of command definitions (all of which implement X Window System calls) are:

```
ACFN2(XDrawlines    ,"IIII2[]II","" )
ACFN2(XOpenDisplay  ,"S"        ,"I"    )
ACFN2(XParseGeometry,"S"        ,"IIIII")
```

As can be seen, not all of the command definition fields need be given explicitly. In the above example, only the first three fields are given explicitly. Instead, they are often set implicitly through the choice of the defining C macro. In the case of `ACFN2` above, the call method is a laid-out argument list, and the two optional parameters are not used. Furthermore, the first argument given to the macro defines both the command name and the C function to be called.

The calls can be grouped into different categories. Each category has a defining C macro associated with it to cut down on the number of items that need to be specified explicitly.

In APL2/X, experience has shown that the commands fall in one of three categories:

1. Most commands can be implemented using the standard facilities available in the base interface. In APL2/X we have implemented about 300 commands this way, or about 75 percent of the total.

2. Some commands require some common pre- or post-processing but are otherwise fairly standard. An example is:

   ```
   AXFAS( XGetGCValues, "IIX", "IG",
          &axGCValues)
   ```

   The call runs the X Window System function `XGetGCValues`. This function returns a pointer to a structure of type `XGCValues`. The cover function takes this pointer and resolves it into its constituent values by using the structure class (`axGCValues`) as a guide to what elements the structure contains. Thus, a call to `XGetGCValues` will return the actual values to APL2, not just a pointer. A number of X Window System calls utilize this function.

   We have used this facility extensively during the development of the interface. A lot of the functionality that is now part of the base interface was prototyped in this fashion and was elevated into the base only when the generality was established.

   In the implementation of the X Window System Xlib calls, 65 calls fell in this category, or 17 percent of the total.

3. The third type of calls consists of the ones that for some reason or another require some specialized pre- or post-processing, e.g.:

```
ACFA0("XNextEvent", "I", "G", axNextEvent)
```

There are a number of reasons why functions end up in this category. Examples are calls returning the X Window System event structures. We use a cover function to convert the event structure pointer to its constituent values, so that the values can be returned by the call, instead of a pointer to the values.

Of the total, 30 X Window System calls required handling as special cases, or about 8 percent.

Command tables. The command definitions are grouped together in tables. For example, all of the X Window System calls are defined in a single table. Typically, a table contains only related commands, although this is not a requirement. These tables form an integral part of the APL2/X interface.

When the *C* or *X* functions are called and the interface gets control from APL2, the interface assumes that the first argument given is the command name. The interface uses this name to search through its command tables looking for a matching command definition. If one is found, it controls any further parameter verification that needs to take place before the actual C function can be invoked.

The default is for the command name matching to be case-sensitive, but it is a matter of a compile-time option to change this default to be case-insensitive.

The command tables not only allowed us to group logically related commands together but also proved to be beneficial during the development stages, where a given set of commands could be worked on by an individual without any fear of overlaying someone else's work. Not all of the tables need be active all of the time; they can be activated and deactivated under user control, and their ordering (governing the command search order) can also be changed.

Currently the APL2/X interface defines the following command tables:

- Xlib calls
- Structure support (structure commands)
- Interface control (system commands)

Tables implementing other collections of C functions and structures can easily be added to this list.

The command tables can also be set up and used in a nested fashion, i.e., subcommands may be specified in a secondary command table. In that case, the command as given by the user is effectively made up of two (or more) separate character strings, one for each command table used. We use this facility to implement some of the APL2/X system commands, but we have not found it that useful overall.

Type codes

The specified command determines what additional parameters need to be given, as well as what information will be passed back as a result of the call. These requirements are described by a series of "type codes" attached to the command, with each parameter described by a single type code.

A large number of type codes have been defined. They are specified using one- or two-character alphanumeric strings. Whenever possible we used the same choice of character codes as those given in *APL2 Programming: System Services Reference* for the APL2/370 processor 11 argument patterns.[12] The code is given on the left side, and its definition follows to its right.

`B1` One-bit Boolean
`B8` Eight-bit unsigned integer
`C1` Character (one-byte)
`E8` Double (eight-byte) floating-point real
`I2` Short (two-byte) integer
`I4` Long (four-byte) integer

Type coercion may be applied by the interface to convert the APL2 data to the type expected by the C function and to convert results from the C function to a type that can be handled by APL2.

To enhance portability we also added definitions that left the actual length of a parameter up to the host environment, e.g.:

`I` Integer—This code can be used whenever a C "int" is called for.

Other additions were called for by specific needs of C and X:

`S` A NULL-terminated character string
`P` A C pointer—This code is treated as a large number that the calling APL2 application program probably should never change.

`X2` Two-byte hexadecimal value
`X4` Four-byte hexadecimal value
`X` Two- or four-byte hexadecimal value, depending on the underlying environment

The hexadecimal values can be specified on input as a bit-vector, as an integer, or as a string of hex characters.

Finally, a couple of special type codes:

`G` Accept any parameter given—This code will often be used where further verification of the input will be performed later. An example can be found in the structure commands. Validation of the content of the structure instance is postponed until the proper structure class definition has been determined.
`_` A place-holder—The value is ignored.

Argument indirection is specified in a C-like manner by prefixing the type code, e.g.:

`*C1` A string of characters
`**I` A double indirect reference to an integer

Arrays are also specified in a C-like manner, e.g.:

`I2[3]` A vector of three (short) integers
`C1[]` A vector of characters—The length is left unspecified, so any length will be accepted.
`I[2;2]` A two-by-two array of integers
`I[2][2]` Another way to specify the above two-by-two array

Some considerations pertaining to arrays:

- Any array passed to a C function is passed as a pointer to the values, not the values themselves, as required by C.
- One or more array dimensions can be left unspecified. The length will then be set according to the incoming data.
- The type code specification is more compact than the one used by the APL2/370 argument patterns and also more like native C and APL2, we believe.

Structures are catered to as well, e.g.:

`{II}` Any combination of type codes can be specified inside the braces, including nested structures.

The type codes are also affected by prefix and suffix modifiers. The prefix modifiers are:

`<` Input only
`>` Output only
`|` Input/output
`?` Optional parameter

The suffix modifiers are:

`...` Repeat last type code as many times as needed to account for the given input values.
`*` Ignore any input parameter beyond those already verified.

Both of these suffix modifiers may only be specified at the very end of a list of type codes or following the last item before a "}" ending a substructure definition.

Parameter passing

All parameters are passed explicitly to and from APL2. APL2/X does not cater to side effects such as update-in-place (i.e., changing the value of an APL2 variable other than by explicit reference), nor does it use call-by-name, where the name of a variable to be used or changed is passed as a parameter and the interface reaches back into the workspace to access the specified variable. Although both are technically feasible to do, there has been neither the need nor the desire to use them. In fact, a conscious effort has been made to stay away from them, as it was viewed as detrimental to the clarity of the resulting code.

On calling a C routine from APL2 only those parameters listed as "input" or "input/output" must be specified (i.e., the parameters listed in the input type code field). The interface will generate whatever output parameter place-holders are needed in the actual call to the C function. Upon completion of the C routine, all parameters listed as "input/output" or "output" will be returned to APL2, in addition to the explicit C function result (if required). We thus take advantage of the ability of APL2 to return multiple values in the explicit result of a function invocation. This is an outgrowth of the desire to avoid relying on (hidden) side effects.

Many X Window System calls return more than one result via their parameters. The parameters used in this fashion are always identified by including the suffix "_return" with the parameter name. These

parameters appear at the end of the parameter list. We have taken advantage of this fact in the way that the input and output type codes are specified in the command definitions.

XParseGeometry is an example of a call returning multiple parameters. The C function prototype and an example of its use via APL2/X are given below:

```
int XParseGeometry(string, x_return, y_return,
                   width_return, height_return)
  char *string;
  int *x_return, *y_return;
  int *width_return, *height_return;

⍝ Multiple output.
      (mask x y width height) ←
         X 'XParseGeometry'  '25×80+10−10'
```

Parameters passed by value

Parameters are passed to and from APL2 by value. This is true no matter what level of indirection is needed by the C routine to be called. The burden of setting up this activity and administering the space is handled by the interface. Thus, using the XParseGeometry example given above, "string" is given as "'25×80+10−10'" in the call from APL2, and the interface will convert this string to the proper "char *" format before calling the real C routine.

Passing the parameters in this fashion maintains the feel of an APL2 function. The housekeeping chores of managing the temporary storage fall upon the interface, not the user.

These statements do not imply that C data pointers are never returned to APL2, or are used by it. Quite the contrary, pointers are typically specified using the "P" or "I" type codes and are passed back to APL2 as large numbers. The application running in APL2 may use this large number on subsequent calls to external functions via APL2/X but will rarely, if ever, have a need to modify the value of the pointer.

Dealing with structures warrants some special comments. We prefer to pass them by value, and given a choice we have set up the calls to do so. However, there are enough exceptions to this procedure to prevent it from being a general rule. The exceptions come about for the following reasons:

1. Performance—It is inherently more expensive in processing time to create the structure on the fly from its values. If a structure instance is being used repeatedly without its content being redefined, it is more efficient to create the structure once and then refer to it using the pointer to the created structure instance.
2. Permanence—The structure may be modified by future calls. It is therefore important that it remains in a fixed location in storage.
3. Hidden side effects—Most X Window System structures do not exhibit this problem, but we did encounter it using the "Xrm" class of calls. Although unstated, the structure pointer was also being referred to in a hidden lookup table. Another manifestation of such effects is where a data structure has an unspecified or hidden prefix or suffix section.

Wherever possible, APL2/X allows structures to be specified on input either by value or by a pointer to an already-existing structure instance.

Support for C data structures

As would be expected of any sizeable C application, the X Window System defines close to 100 C data structures. Therefore, to fully support the X Window System, the APL2/X interface had to be able to provide access to these data structures as well as the many function calls that are defined by the X Window System. In doing so, APL2/X has implemented these data structures in C and provided import and export access from APL2.

Those familiar with the object-oriented paradigm will recognize the similarities in that approach to the APL2/X handling of data structures. APL2/X maintains a structure (class) definition as part of the C command interface. APL2 calls upon this definition to create new instances of the structure in memory and to assign values to and retrieve values from the fields (class data members) of the instance.

The structure instances are stored in memory controlled by C and thereby directly available to the C application, in this case the X Window System. Upon request from APL2, the instance of the data structure is mapped to an APL2 vector. The vector may be simple (homogeneous) or general (heterogeneous), depending on the underlying C definition. When in APL2, the array can be manipulated in the normal APL2 fashion.

Structure commands. A common set of structure commands has been defined to allow APL2 to easily

create and access the data structure instances maintained by C. Again one can draw comparisons to these structure commands and those implemented for class definitions in many object-oriented languages. The structure commands provide the means to create instances of a given structure type, to perform the chores of getting data in and out of it, and to free up the space once it is no longer needed.

Listed below are the commands that are defined. The commands are shown in three groups: those in the left column operate on a single instance of a structure, the commands in the middle column operate on multiple adjoining structures, and the ones on the right return assorted information from the structure definition.

```
Clear          MClear          GeConst
Get            MFree           GetFields
New            MGet            GetSize
Put            MNew
NewPut         MPut
SFree
```

Structure command usage. The syntax common to all of the structure commands includes the command name followed by the structure type. For those commands that deal with existing structure instances, the pointer to the structure instance (its handle) is expected as the third argument. Following is the general structure command syntax as called from APL2:

```
(rc [result]) ← C command struct [parm] ...
```

Some examples of using these commands are:

```
⍝ Create a new XTextItem instance
      item ← X 'New' 'XTextItem'

⍝ Now fill it with data
      X 'Put' 'XTextItem' item
         ('Simple' 1 2 3)

⍝ Verify that the data made it in
      X 'Get' 'XTextItem' item
 Simple 1 2 3

⍝ Use the structure in a call
      X 'XDrawText' dp w gc x y item 1

⍝ Remember to free it when all done
      X  'SFree' 'XTextItem' item
```

Structure type definitions. The structure type definitions are grouped in tables in the same manner as are the command definitions. In fact, the APL2/X interface provides for each environment grouping to accommodate both a command table and a structure table, as these definitions often go hand in hand. Currently, these three structure definition tables are provided by APL2/X: X events and other X structures, and C primitive structures.

In order to have the structure commands work, the tables must specify the structure type being addressed. The structure type located in one of the predefined tables provides the definition of the elements of a structure instance of that type. Specifically, the type definition contains information about each field of the structure, the names, and data types. The field data types are specified using the same type codes as are used for the function arguments in the command tables.

This structure definition information is also readily available from APL2 via the interface. Having this information available can be of great assistance when using the data structures from within APL2, in that it associates each element in the vector with its related field name in C.

To help illustrate the point, this is how the `XTextItem` structure from the X Window System Xlib.h header file is defined in C:

```
typedef struct {
  char *chars;  /* Pointer to string        */
  int   nchars; /* Number of characters     */
  int   delta;  /* Delta between strings    */
  Font  font;   /* Font to be used, or None */
} XTextItem;
```

APL2 accesses the structure information in the following manner:

```
⍝ Get all XTextItem fields
      X 'GetFields' 'XTextItem'
 char *chars S
 int nchars  I
 int delta   I
 Font font   I
```

Note that the full C definition of the field is maintained even though the field name and the field type code are the only pieces of information used by APL2/X. The C data type specification (e.g., char *, int, or Font) is kept as part of the field definition since it is often very useful, if not crucial, to the understanding of the role of a given structure member.

Structure field access. To accomplish the equivalent of field access by name as provided by C, two APL2 functions, *axGetFF* and *axGetFF1*, are included as part of APL2/X. These functions use the field information provided by *GetFields* to associate indices to the various field names, thereby providing the index-by-name capability for a related structure instance held in an APL2 vector. The main difference between these two functions is that *axGetFF* provides the indexing for all of the fields in the structure, and *axGetFF1* returns index information for selected fields specified in the call.

By means of an example, we now demonstrate how the `chars` field of an `XTextItem` structure instance, `text`, is accessed from both C and APL2. Note that the *axGetFF* function has previously been called in APL2 to associate the correct index to the field name:

In C:	In APL2:
`text.chars`	`text[chars]`
`text->chars`	`text[chars]`

As a benefit of obtaining the field indexing of the structure from C, the APL2 application can have a measure of independence from changes in the order of fields in the underlying C data structure. That is to say that as long as the fields remain intact and the C structure definition is maintained in accordance with the C application, the APL2 application will not have to change either.

Abandoned approach. Originally we implemented the structure support using "typed" instances so that each instance had a hidden header section that identified the structure type. This implementation meant that the structure class did not have to be specified on each structure command since the information was already available. However, when the structure was allocated by the C application instead of the APL2/X interface, it meant a lot of extra work because the interface would have to allocate another instance with the proper header attached and then copy the structure data of the application into this new area. With the implementation of nested structures this activity became difficult to control, so we ultimately abandoned the "typed" instance approach.

Support for C constants

If the X Window System defines a large number of structures, it defines ten times that many constants in its header files. As any experienced programmer would attest, the use of constants is a major benefit to an application in that it provides symbolic reference so that when a change is called for, only the constant value needs to be changed, regardless of how many references exist. Because these constants disappear during the compilation process, there is no penalty for defining large numbers of them, and the X Window System takes advantage of this and defines a large number of these constants in its header files.

The sheer number of constants employed by the X Window System dictated that APL2/X implement access to these constants in a selective manner rather than expose the whole lot. This approach is logical since any given constant is typically used by only a very limited number of structures or functions. In fact, in the majority of cases, the constants defined in the X Window System header files are related to specific fields of a structure. Therefore, in giving APL2 access to these constants, the constants are logically tied to a related structure definition.

The `GetConst` command provided by APL2/X as part of the structure commands is used to retrieve the constant values associated with a given structure for use in APL2. Following is an example of the output from this command:

```
      X          'GetConst'      'XSizeHints'
USPosition           X                 1
USSize               X                 2
PPosition            X                 4
PSize                X                 8
PMinSize             X                16
PMaxSize             X                32
PResizeInc           X                64
PAspect              X               128
PBaseSize            X               256
PWinGravity          X               512
PAllHints            X               252
```

It is a simple task for an APL2 function to issue this call, create a set of variables, and initialize them to the constant values that are returned. In fact, the *axGetFF* and *axGetFF1* functions previously introduced in the last section not only define structure field indexing, they also create these constant variables for use by the APL2 application.

By doing so, the APL2 functions are able to use the same constants as defined by the X Window System. Such usage insulates the application from changes to these constant values. We experienced

an example of this when upgrading the APL2/X interface support of the X Window System from release 11.3 to release 11.4. Release 11.4 had changed some of the constants associated with the `XSizeHints` structure, among other changes. These changes meant that the table holding the constants in APL2/X had to be recompiled to pick up the changed values, but through the use of the *axGetFF* function it never affected the APL2 applications.

System commands

APL2/X provides a group of system commands in addition to the structure and X Window System commands. These commands are used to control and interrogate the interface itself, as opposed to accessing and using external functions that supply the application with needed services. The names of these commands all start with a closing parenthesis, mimicking the APL2 system commands.

The following system commands are presently defined:

`)Cmds`	List the available commands
`)Env Get`	Get current command environments
`)Env Set`	Change the command environment order
`)RC`	List a return code message
`)Structs`	List the available structures
`)Syntax`	List the syntax of a specific command
`)Version`	Return the APL2/X version identifier

Some examples of their use follow:

```
      X ')Syntax' 'XParseGeometry'
Xlib  XParseGeometry  S  IIIII

      X ')Version'
APL2/X Development Version 0.00

      X ')Env' 'Get'
Xlib  Structs  System
```

Return codes

A major difference between APL2/X and processor 11 of APL2/370 is in the way that errors are reported. Processor 11 treats this condition at an atomic level, using the normal APL2 error messages such as *DOMAIN ERROR* and *VALUE ERROR*. If the error stems from using an element of the wrong type in a vector of arguments, it can be quite difficult to locate the source of the error, especially since the APL2/370 *⎕NA* argument pattern information is not directly available to the application.

APL2/X improves error reporting in several ways. First of all, the arguments in error can easily be determined, since each argument passed to APL2/X will be associated with a return code. Second, the return code is tied to an error message explaining the source of the error, if using the *X* APL2 function. Third, the syntax of the call is available for inspection via a system command.

For instance, using the *X* function:

```
      X 'XOpenDisplay'
Error in input (RC=1)
 Index rc parm
     0  0 XOpenDisplay
     1 16
 16 Expected parameter of type '%s' is missing
Command 'XOpenDisplay' defined by:
   Xlib XOpenDisplay SI
      X 'XOpenDisplay'
      ^
```

Note the use of the default error handler that is part of the *X* function; it will halt execution at the place of error and will point out the parameter or parameters in error.

Using *C* instead (without the trailing comment, of course; it is just placed here for explanation):

```
      C 'XOpenDisplay' 'first' 'second'
 17 0 0  17
⍝ 17: Too many parameters
```

The *C* function does not halt the processing when an error is encountered. Instead, it returns a nonzero return code to the application, and it is up to the application to take whatever corrective action is required. Note the structure of the element return codes: it contains an element for each given or required parameter, whichever count is the larger of the two. This way it is possible to uniquely identify the source of any errors in the parameters.

This principle extends to nested parameters as well, as the following example shows:

```
      C 'Put' 'XTextItem' 509120
        (1 'text' 2 3 4)
 1    0 0 0  26 21 0 0 17
⍝ 17: Excessive number of parameters given
⍝ 21: Dimension 90i must be equal to 90s
⍝ 26: Cannot convert from type 90 to type 90s
```

Issues in calling C routines from APL2

The initial version of APL2/X was completed on a VM system, using processor 11 of APL2 Version 1 Release 3 to call functions external to APL2 itself. The only two programming languages specifically mentioned in the documentation for processor 11 are FORTRAN and System/370 Assembler.[13] Initially we used the FORTRAN linkage-type of processor 11 rather than OBJECT. It was chosen because it would include the length information for each parameter passed. However, in trying to call routines written in C, we encountered the following problems that had to be solved in order for us to implement the X Window System interface:

- Character strings not null-terminated—Character strings are by definition required to be terminated by a null byte in C, but processor 11 does not ensure that the strings passed are null-terminated.
- Returning the result of a C function to APL2—C functions compiled with the C/370 compiler place the result in register 1, but processor 11 expects a result to be passed back to APL2 in register 0.
- Using C pointers—It is not possible to specify a given parameter as being a pointer, such as the C definition `char *` would require. The argument patterns[14] of processor 11 do not cater to this type of definition, and it is therefore possible to handle the distinction of passing a parameter by value, as opposed to passing it by reference.
- Fully specified function argument patterns—The function argument pattern of processor 11 must be completely known by the time a function is called. It is not possible to defer processing and verification of some of the arguments until later, or to ignore others altogether. Thus, it is not possible to call a given function with differing types of arguments.

The above problems are related to calling a single C function. In addition to these problems, trying to implement an X Window System interface introduces another set of problems related to the sheer number of calls to support (395 in the case of the X Window System):

- No list options—There is no call to obtain the function argument pattern of a given external function from within APL2 (short of extracting it from the names file), or to obtain a list of all the accessible external functions.
- Cumbersome to implement and maintain—For a function to be used, it must have an entry in both the names file and the assembler stub module, as well as a *⎕NA* definition in the workspace. Each workspace needing access to the X Window System therefore ends up with a large number of function definitions, in most cases swamping the real functions of the application.

As can be seen, most of these problems revolve around parameter passing. They have been solved in APL2/X by having the interface itself take over the parameter verification chore, using the FUNCTION linkage-type of processor 11, without any parameter verification imposed by the processor. And instead of storing the argument patterns in an external names file, APL2/X now stores these patterns in command tables internal to the interface. Thus, what APL2/X receives is the APL2 data specified by the calling function, and it is up to the interface to perform any needed parameter validation and coercions. This scheme has given APL2/X maximum control of the parameter passing, and thus the following results have been achieved:

- Only a single external function is established in the workspace. The name of the C function to be called is now passed as the first argument in the call.
- Null-termination of character strings is handled automatically by the interface. It avoids having the caller do it in APL2 by either imposing a fixed-length restriction on each string or requiring that the string include the `NULL` terminator.
- The interface supports pointer variables. The support caters to an unlimited number of reference indirections. As an example, an argument with a declaration of "`int **`" is supported. This would be specified as "`**I`".
- Argument verification has been extended to allow for deferred verification. Such verification has proved to be especially important when working with data structures, where the content and structure can vary greatly from structure to structure.
- Additional data types are supported, such as hexadecimals. Also, some data types can be specified in multiple ways. An example of the latter is a bit-field, which can be specified as a vector of bits, an integer (i.e., packed bits), or in hexadecimal format (in the form of a character string).
- Multiple results can be returned as explicit results of the call to a given C function, without the need to build special APL2 functions that preallocate variables to hold the returned information.

- Commands have been added to interrogate the interface itself. This interrogation enables the interface to return the expected syntax of a given call or provide lists of the commands and structures supported.
- Using function linkage has enabled APL2/X to use the processor 11 service routines. These routines provide some useful services, such as data conversion and execution of APL2 expressions from within C.
- A large number of utility functions have been implemented in C that allow us to process and manipulate APL2 data structures in C in an easy and proficient manner.

Taking over the argument verification job turned out to be a blessing in disguise for APL2/X. It made the "port" to the APL2/6000 and APL2/PC environments very easy to accomplish. (In APL2/PC, only the basic APL2-to-C interface has been implemented, not the support for the X Window System.) Both of these environments communicate with APL2/X via a shared variable interface, unlike the APL2/370 implementations. Except for different internal formats of the APL2 data passed from APL2, the processing remains the same as far as APL2/X is concerned, at the internal level and, more importantly, at the user interface level too.

Changes to the X Window System call syntax

One of the design goals for APL2/X was to implement as faithful a representation of the X Window System in APL2/X as possible. However, some differences exist due to the very different nature of C and APL2. The important differences are:

- Function arguments are always specified by value in the same way that they would be for regular APL2 functions. This is true for all types of arguments, scalars as well as arrays. APL2/X performs any needed type coercion and also adds any required indirection pointers based on the type code information before making the actual call to the C routine.

  Note that the explicit use of pointers in APL2/X is not precluded. In fact, they are used as such in many of the X Window System calls, as well as in the routines that implement the structure calls. In these cases, the pointer given is a "magic" constant; as far as the application is concerned it is a value that uniquely identifies some available resource, and no explicit changes to the value should be attempted. A prime example of such a constant is the X Window System "display pointer." This pointer is used on most X Window System calls, but no calculations are ever performed on the pointer itself.

- Only input parameters may be specified on the calls to the X Window System. APL2/X automatically adds any needed output parameters that the call may require. It is a change from the C environment, where the output parameters must be specified explicitly on the call and space possibly allocated to hold the results.

- All results from calling a function are returned as explicit results, including results returned in C via changes to the output arguments. No side effects such as changing of global variables in the workspace are employed. Also, pointer arguments are de-referenced, so what is returned in APL2 are the data values, not the pointers.

The above differences are a consequence of the basic design philosophy underlying the APL2-to-C interface. Another difference, described next, is specific to the X Window System calls dealing with event structures and is more a matter of convenience.

X Window System events are always set or returned by value. The event data are then immediately available for use in the APL2 environment, instead of the structure commands being employed to retrieve the event structure values on the basis of a returned pointer value. The rationale for this decision is that the event data are almost invariable as required by the APL2 application, not just the event pointer, so APL2/X returns the data to speed up the process. In the rare cases where the event pointer is required, it can be acquired through a separate, special call to the interface.

Potential improvements

Although we have come a long way in providing APL2 with access to the X Window System, more work can certainly be envisioned. First among the possibilities would be to add a layer of APL2 functions to help use the X Window System facilities. This could shield some of the complexity of the X Window System, in much the same way it was done in the past in the workspace FSC126 that helped APL2 create and use a full-screen panel by accessing routines of GDDM via APL2 cover functions.

A second option is to extend the range of supported X Window System routines. The Xlib layer is the only layer supported today. There appear to be no technical problems in extending the support to higher layers of the X Window System functionality. It is certain that APL2 would benefit from gaining access to higher-level routines that create and manipulate window system items such as menu bars, sliders, pop-up windows, and other items associated with a modern, windowed user interface. Indeed, this possibility is not restricted solely to the X Window System libraries; other collections of C functions can be accessed equally well from APL2 via this interface.

In an effort to improve the interface for use with C applications in general, some experimental work has already gone into providing the means to dynamically define C commands and structures from APL2. This capability allows APL2 to directly interface with existing C applications without requiring the definitions of the related functions and structures to be built into the APL2/X interface itself.

Last, an even better support for data structures is possible if implemented in APL2 itself, maybe in the form of an option on □*NA* to allow APL2 to access external data variables in much the same way that external functions today are supported. An advantage would be a single copy of data, with the obvious corollary of improved data integrity.

A final example

It would appear as though it is a rite of passage for a windows-based system to have a "HelloWorld" sample program. APL2/X follows this trend. The HelloWorld APL2 function listed in Appendix A illustrates how many of the concepts and ideas presented in this paper fit together. It shows how an APL2 function can implement the two fundamental concepts of a windows-based system: window manipulation and responding to user-generated events. We will let the function listing speak for itself as to the detail; for a more in-depth discussion of the program, see *Introduction to the X Window System*, Chapter 2,[15] *IBM AIX APL2/6000 User's Guide*,[16] or *An Interface Between APL2 and the X Window System*.[17]

Note that the function as listed takes a simplistic view of the world. It has only minimal error-checking, and it is coded as a single, large function. A production-level version of the same function would certainly have to do a more thorough job verifying that error conditions had not occurred. Also, much of the functionality would be implemented through secondary functions common to many windowed applications. However, since the focus of this paper is purely and solely on the capability to access and call C and the X Window System routines from APL2 functions, this example is presented in the form given.

Summary

A major goal achieved in this project was to enable APL2 to use the exciting new facilities that the X Window System embodies and to bring to the X Window System the power of the APL2 interactive environment and array-handling capabilities. This truly brings the potential of a modern-day interface to APL2 while at the same time augmenting the X Window System. A second goal was to provide a common interface to the C language from all of IBM's APL2 systems, ranging from PC DOS through AIX on the RISC System/6000 to VM and MVS, including full support of C data structures. A third goal was to maintain the function-call "feel" of APL2, enabling the external functions to be used as though they were truly written in APL2.

To achieve these goals a number of large issues had to be overcome. Among the more daunting ones were data mapping, handling storage management, and automatic parameter indirection so vital to any C interface. Since APL2 and C are so diverse in the way they deal with storage management, it proved to be a real challenge, especially when dealing with data structures.

The APL2/X interface described is currently available to IBM customers on two APL2 platforms. In APL2/6000 for AIX on the RISC System/6000 (Program No. 5765-012) it is included as the AP144 auxiliary processor,[18] and it is provided as a sample offering with TCP/IP Version 2 for VM (Program No. 5735-FAL) to be used by APL2/VM.[19]

Acknowledgments

We would be remiss if we did not acknowledge the significant amount of help we have received from many people. A special acknowledgment goes to Bob Cohen, who worked with us part time while pursuing his Ph.D., for implementing and verifying a good portion of the X Window System calls. We would also like to thank our manager, Love Sea-

wright, and our center manager, Dick MacKinnon, for making it possible for us to engage in this project; Andy Pierce for showing us his REXX-based interface and providing us the X Window System on VM/CMS; Mike VanDerMeulen and John Mizel for helping us port the interface to the RISC System/6000; Ray Trimble, Michael Wheatley, Nancy Wheeler, and David Liebtag for helping us through the □NA of APL2; Elbert Hu for including APL2/X with TCP/IP Version 2 for VM; and the many people that answered our queries on the electronic forums and provided us with good feedback during the development.

* Trademark or registered trademark of International Business Machines Corporation.

** Trademark or registered trademark of Massachusetts Institute of Technology, UNIX System Laboratories, Inc., Microsoft Corporation, or Open Software Foundation, Inc.

Appendix A: HelloWorld listing

```
[0]    HelloWorld;□IO;dp;w;gc;s;e;k;rw;bp;wp;m
                 ;hello;hi;done;None;hp;hints;rc
                 ;nl;x;ep
[1]    ⍝ Sample X program, based on helloworld.c
[2]    ⍝ from Oliver Jones:
[3]    ⍝ Introduction to the X Window System
[4]    ⍝ Prentice-Hall, 1989;  ISBN 0-13-499997-4
[5]     □IO←0
[6]
[7]    ⍝ Define some constant text-strings
[8]     hello←'Hello, World.'
[9]    ⍝ The exclamation point makes hi ugly:
[10]    hi←'Hi',('A'=□AF 65)⊃□AF 90 33
[11]
[12]   ⍝ Initialization
[13]    →(0=dp←X 'XOpenDisplay' '')↓lopen
[14]    □←'XOpenDisplay failed ...'
[15]    □←'... HelloWorld aborted'
[16]    →lexit
[17]   lopen:
[18]
[19]   ⍝ Default pixel values
[20]    s←X 'XDefaultScreen' dp
[21]    bp←X 'XBlackPixel' dp s
[22]    wp←X 'XWhitePixel' dp s
[23]
[24]   ⍝ Define an X constant
[25]    None←0
[26]
[27]   ⍝ Prepare to set window position and size
[28]    (rc nl)←'H' axGetFF 'XSizeHints'
[29]    m←↑/PPosition PSize
[30]
[31]   ⍝ Build an XSizeHints structure instance
[32]    hp←X 'New' 'XSizeHints'
[33]    hints←X 'Get' 'XSizeHints' hp
[34]    hints[H_flags H_x H_y]←m 200 300
[35]    hints[H_width H_height]←350 250
[36]    X 'Put' 'XSizeHints' hp hints
[37]
[38]   ⍝ Window creation
[39]    rw←X 'XDefaultRootWindow' dp
[40]    x←hints[1 2 3 4],5 bp wp
[41]    w←X 'XCreateSimpleWindow' dp rw,x
[42]    x←hello hello None('A' 'test')2 hp
[43]    X 'XSetStandardProperties' dp w,x
[44]    X 'SFree' 'XSizeHints' hp
[45]
[46]   ⍝ Create a Graphics Context
[47]    gc←X 'XCreateGC' dp w 0 0
[48]    X 'XSetBackground' dp gc bp
[49]    X 'XSetForeground' dp gc wp
[50]
[51]   ⍝ Window mapping
[52]    X 'XMapRaised' dp w
[53]
[54]   ⍝ Input event selection
[55]    m←'ButtonPressMask' 'KeyPressMask'
[56]    m←m,∈'ExposureMask'
[57]    (rc m)←m axGetFF1 'XEvent'
[58]    X 'XSelectInput' dp w(+/m)
[59]    ep←X 'XGetEventBuffer'
[60]
[61]   ⍝ Get some more constants
[62]    m←'KeyPress' 'ButtonPress'
[63]    m←m, 'Expose' 'MappingNotify'
[64]    (rc m)←m axGetFF1 'XEvent'
[65]   ⍝ ... and some event structure layouts
[66]    nl←nl,1⊃'K_' axGetFF 'XKeyEvent'
[67]    nl←nl,1⊃'B_' axGetFF 'XButtonEvent'
[68]    nl←nl,1⊃'E_' axGetFF 'XExposeEvent'
[69]
[70]   ⍝ Main event-reading loop
[71]    done←0
[72]   levent:→(done=0)↓lend
[73]
[74]   ⍝ Read and process the next event
[75]    x←1KeyPress 1ButtonPress
[76]    x←x,1Expose 1MappingNotify
[77]    →(m=K_type⊃e←X 'XNextEvent' dp)/x
[78]
[79]   1Expose: ⍝ Repaint window on expose events
[80]    →e[E_count]↑levent    ⍝ Count > 0 ?
[81]    x←e[E_display E_window],gc,50 50
[82]    X(∈'XDrawImageString'),x,hello(⍴hello)
[83]    →levent
[84]
[85]   1ButtonPress: ⍝ Process mouse-button presses
[86]    x←e[B_display B_window],gc,e[B_x B_y]
[87]    X(∈'XDrawImageString'),x,hi(⍴hi)
[88]    →levent
[89]
[90]   1KeyPress: ⍝ Process keyboard input
[91]    k←∈1⊃X 'XLookupString' ep
[92]    →(done←(↑k)∈'qQ')↑levent
[93]    x←e[K_display K_window],gc,e[K_x K_y]
[94]    X(∈'XDrawImageString'),x,k(⍴k)
[95]    →levent
[96]
[97]   1MappingNotify: ⍝ Reset keyboard
[98]    X 'XRefreshKeyboardMapping' e
[99]    →levent
[100]
[101] /end: ⍝ Termination
[102]   X 'XFreeGC' dp gc
[103]   X 'XDestroyWindow' dp w
[104]   X 'XCloseDisplay' dp
[105]   nl←□EX'nl
[106] lexit:
      ∇  1991-4-16 18.43.0 (GMT-4)
```

Cited references

1. J. A. Brown, S. Pakin, and R. P. Polivka, *APL2 at a Glance*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1988).
2. O. Jones, *Introduction to the X Window System*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).
3. D. Comer, *Internetworking with TCP/IP*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1988).
4. *IBM Transmission Control Program/Internet Protocol Version 2 for VM: Programmer's Reference Manual*, Appendix A, SC31-6084, IBM Corporation (1990); Program No. 5735 FAL; available through IBM branch offices.

5. *IBM Transmission Control Program/Internet Protocol Version 2 for MVS: Programmer's Reference Manual*, SC31-6087, IBM Corporation (1991); Program No. 5735 HAL; available through IBM branch offices.
6. *X3270—AIX X Windows 3270 Emulator User's Guide*, SC23-0579-0, IBM Corporation (1991); available through IBM branch offices.
7. J. A. Pierce and R. O. Reynolds, *The X Window System in the S/370 Environment*, G325-4100-0, IBM Corporation (1991); available through IBM branch offices.
8. *APL2 Programming: Processor Interface Reference*, SH20-9234-0, IBM Corporation (1987), pp. 15–23; available through IBM branch offices.
9. *APL2 Programming: APL2 for the IBM PC, User's Guide, Version 1.02*, SC33-0600-2, IBM Corporation (1990), pp. 436–438; available through IBM branch offices.
10. R. W. Scheifler and J. Gettys with J. Flowers, R. Newman, and D. Rosenthal, *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFS*, Digital Press, Bedford, MA (1990).
11. *AIX Calls and Subroutine Reference for RISC System/6000, Volume 4: User Interface*, SC23-2198, IBM Corporation (1990); available through IBM branch offices.
12. *APL2 Programming: System Services Reference*, Chapter 23, SH20-9218, IBM Corporation (1990), pp. 238–239; available through IBM branch offices.
13. *Ibid.*, p. 237.
14. *Ibid.*, p. 243.
15. See Reference 2, Chapter 2.
16. *AIX APL2/6000 User's Guide*, SC23-3051-0, IBM Corporation (1991), pp. 305–314; available through IBM branch offices.
17. *An Interface Between APL2 and the X Window System*, IBM licensed material provided with TCP/IP Version 2 for VM (Program No. 5735-FAL), pp. 5–14; available through IBM branch offices.
18. See Reference 16, pp. 209–216.
19. See Reference 17.

Accepted for publication June 10, 1991.

John R. Jensen *IBM Cambridge Scientific Center, 101 Main Street, Cambridge, Massachusetts 02142.* Mr. Jensen is a scientific staff member at the Cambridge Scientific Center. He joined IBM Denmark in 1978 as a systems engineer. In 1982, he worked at the IBM Canada Computing Centre in Vancouver, British Columbia, and from 1983 to 1988 was at the Dallas Development Laboratory in Texas, working on the IC/1 and OfficeVision products. He became a member of the Cambridge Scientific Center staff in 1988. His current areas of interest include user interface design, application prototyping, programming environments, and compilers. Mr. Jensen received an M.Sc. degree in electrical engineering from the Technical University of Copenhagen, Denmark, in 1978 and an M.B.A. in accounting from the Copenhagen School of Economics in 1981. He is a member of the ACM and the IEEE Computer Society.

Kirk A. Beaty *IBM Cambridge Scientific Center, 101 Main Street, Cambridge, Massachusetts 02142.* Mr. Beaty is a scientific staff member at the Cambridge Scientific Center. He joined IBM at Sterling Forest, New York, in 1981 as a systems programmer. Furthering his experience at Sterling Forest, from 1983 to 1987 he became involved in telecommunications, including the technical software leadership role in the creation of IBM's centrally managed internal VNET backbone network. He has been a member of the Cambridge Scientific Center since 1987. Mr. Beaty received a B.S. with honors in mathematics/computer science (while minoring in business administration) in 1981 from Manchester College, North Manchester, Indiana. He is a graduate of the IBM Systems Research Institute and has recently completed a Certificate of Advanced Study in software engineering at Harvard University.

Reprint Order No. G321-5447.