DECwindows Without C

Jeremy Begg
VSM Software Services
Adelaide, South Australia

Abstract

Almost all programs written for the X11 Window System are written in C. However, many programmers are either unfamiliar with C or prefer not to use it, and not every site has a C compiler. With Digital's DECwindows under VAX/VMS the programmer has the option of writing X11 programs in languages other than C using reasonably standard VAX/VMS calls. This paper presents the author's attempts to write a simple application using VAX Pascal. Some familiarity with DECwindows programming (including UIL) is assumed but a knowledge of Pascal is not required.

Introduction

DECwindows is Digital's supported version of the X11 Window System (commonly referred to simply as "X"), developed at the Massechusetts Institute of Technology (MIT) over several years. DECwindows is available for VMS and ULTRIX systems and the X11 Server component is available for MS-DOS and Macintosh computers with the PATHWORKS product.[1]

Many computer vendors supply only the standard MIT distribution of X, if they supply one at all, and programmers on those systems are restricted to writing the X11 programs in C using the standard MIT-defined data structures and routines.

Under VMS and ULTRIX, programmers have the option of using many of the languages which Digital supports on these operating systems. This paper concerns itself only with VMS applications. Programmers writing for VMS have the choice of the standard MIT C interface to X11 or can use the so-called VMS Bindings, which use routines and data structures written to the VAX procedure calling standards. The VMS bindings are very similar to the MIT C bindings and interface to the standard X11 library calls.

In the DECwindows programming documentation, each DECwindows routine is described in two ways, one for the standard MIT C binding and one for the VMS binding. Figure 1, an extract from the "VMS DECwindows Toolkit Routines Reference Manual", shows the doubly formatted description. However, much more information about the routine is needed before it can be used. The argument information lists the "result" parameter's data type as unsigned longword, but that is not necessarily the parameter's formal declaration in the relevant definition file.

Files Used in DECwindows Programming

The DECwindows definition files for all languages are in two directories, **DECW$INCLUDE:** and **SYS$LIBRARY:**.

DECW$INCLUDE: is the location of the `.h` files for programming in C using the MIT binding, and also a selection of .UIL files (which are the same for all languages). In particular, the file **DWTAPPL.UIL** contains definitions of widget arguments etc used in UIL files.

SYS$LIBRARY: is where definition files for all languages can be found, when programming with the VMS binding (including C). Also, the shareable images which contain the DECwindows routines are found here.

The files in SYS$LIBRARY are roughly organised by the sophistication of the application being programmed. The more commonly used files, for all languages except ADA, are as follows:

- DECW$DWTDEF.*
  Structures and routines for using the DECwindows Toolkit

[1] VMS V5.5 is the last major release of VMS to include DECwindows as standard. The next major release will require the customer to purchase DECwindows/Motif. This paper is written assuming that vanilla VMS DECwindows is in use, not DECwindows/Motif, but the principles remain the same.

- DECW$DWTENTRY.*
  Routines in the DECwindows Toolkit
- DECW$DWTSTRUCT.*
  Structures for using the DECwindows Toolkit
- DECW$DWTWIDGETDEF.*
  Structures and routines for using the DECwindows Toolkit, plus extra definitions for dealing with widget internals
- DECW$DWTWIDGETSTRUCT.*
  Structures used in dealing with with widget internals
- DECW$DWTMSG.*
  VMS error message code definitions
- DECW$XLIBDEF.*
  Structures and definitions for using the Xlib
- DECW$XLIBMSG.*
  VMS error message codes for Xlib calls

Programs which don't make explicit reference to Xlib routines or data structures will not need to include the DECW$XLIB* files.

When programming in Ada, there are only two files to include:

- DECW$DWT_.ADA
  Contains <u>all</u> DECwindows Toolkit defnitions

FIGURE 1: Documentation page for the DECwindows Toolkit routine **PENDING**

PENDING

Determines if any events are pending on the input queue.

VAX FORMAT

result = **XT$PENDING** ()

Argument Information

Argument	Usage	Data Type	Access	Mechanism
result	Boolean	uns longword	write	value

MIT C FORMAT

result = **XtPending** ()

Argument Information

```
XtInputMask XtPending()
```

RETURNS

result

A nonzero value if there are events pending from the server, timer or other input sources. The **result** argument is a bit mask that is the OR of XtIMXEvent, XtIMTimer, and XtIMAlternateInput. (The symbolic name XtMAll is the bit OR of all event types.)

Description

PENDING returns a nonzero value if events are pending from the server or from other input sources in the defaults application context. If not events are pending, PENDING flushes the output buffer and returns zero.

INITIALIZE must be called before using this routine.

- DECW$X_.ADA
  Contains all Xlib definitions.

For BASIC, BLISS, C (VMS binding), FORTRAN and PL/1, the DECW$DWTDEF file contains only two statements, to include the DECW$DWTENTRY and DECW$DWTSTRUCT files. Similarly, the file DECW$DWTWIDGETDEF includes these two files plus the DECW$DWTWIDGETSTRUCT file.

Most programmers will use DECW$DWTDEF or DECW$DWTWIDGETDEF.

Linking a DECwindows Application

A linker options file is commonly used when linking DECwindows applications, as the link process must access one or more shareable images to resolve the DECwindows references. The linker options file will be the same regardless of which language or binding is being used, and will most likely contain one or both of the following statements:

```
! To link the DECwindows Toolkit
SYS$SHARE:DECW$DWTLIBSHR/SHARE
! To link the Xlib routines
SYS$SHARE:DECW$XLIBSHR/SHARE
```

Other commonly used shareable images in SYS$SHARE which might have to be added to the options file include:

- DECW$PRINTWGTSHR.EXE
  The DECwindows Print Widget
- DECW$TERMINALSHR.EXE
  For creating DECterms

Thus the link command for a program making DECwindows Toolkit calls, including the print widget, and creates a DECterm, would be something like:

```
$ LINK program, DECW/OPT, ...
```

where DECW.OPT includes the lines

```
SYS$SHARE:DWTLIBSHR/SHARE
DECW$PRINTWGTSHR/SHARE
DECW$TERMINALSHR/SHARE
```

A "Hello World" Program

The first program anyone writes is traditionally "Hello World" and DECwindows is no different. Here are two versions, one using the MIT C binding and one written in Pascal using the VMS binding.

First, the C version:

```
#include <decw$include:intrinsic.h>
#include <decw$include:stringdefs.h>
#include <decw$include:dwtwidget.h>

main (argc,argv)
int argc;
char **argv;
{
  Widget topLevel, hello;

  topLevel = XtInitialize(argv[0]
                          ,"XHello"
                          ,NULL
                          ,0
                          ,&argc
                          ,argv
                          );

  hello = XtCreateManagedWidget("hello"
                          ,labelWidgetClass
                          ,topLevel
                          ,NULL
                          ,0);

  XtRealizeWidget(topLevel);

  XtMainLoop();

}
```

And the Pascal version:

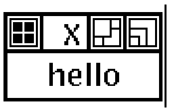
```
[inherit
('SYS$SHARE:DECW$DWTWIDGETDEF')
]
Program XHello (input,output);

Const
  progname = 'XHello';

Var
  topLevel, hello: Dwt$Widget;
  arglist: array [1..1] of Dwt$Arg;
  argcount: Dwt$Cardinal := 0;

Begin

{ Initialize the X11 environment }
topLevel := Xt$Initialize (progname
                          ,progname
                          ,0
                          ,0
                          ,argcount
```

```
                                      ,
                                      );


hello := Xt$Create_Managed_Widget
      ('hello'
      ,labelwidgetclass::Dwt$Widget_Cls
      ,topLevel
      ,arglist
      ,argcount
      );

Xt$Realize_Widget (topLevel);

Xt$Main_Loop;

end.
```

In C, `argv[0]` contains the program name, but Pascal does not provide a mechanism for this, so the program name is passed explicitly. In this example, the name is hard-coded into the Pascal program, but it would be quite feasible to use SYS$GETJPI to determine it at run time.

Similarly, C provides its own way of extracting command line arguments, which X11 supports, but Pascal does not so the corresponding parameters to **Xt$Initialize** are not used in the Pascal program.

The last two parameters to **Xt$Create_Managed_Widget** are used to specify non-default attributes for the new widget. The default text for a label widget is the name of the widget; the end result looks like this:



Some Problems with the VMS Bindings

Even such a small program as "Hello World" is enough to hint at problems to come when using a language other than C. In the Pascal program's call to **Xt$Create_Managed_Widget** it was found necessary to type-cast the widget class parameter because the relevant definitions in the Pascal definition file, SYS$SHARE:DECW$DWTWIDGETDEF.PEN, are not quite compatible: `labelwidgetclass` is a variable of type `Label_Class`, which is not the same as type `Widget_Class`.

Incompatibilities such as this paragraph occur frequently enough to be annoying, but not so frequently as to be totally discouraging! It is suspected that the various definition files were built automatically from SDL files based on the MIT C bindings, but without sufficient controls to ensure the files are readily usable. This is particularly poor in a language like Pascal which is quite strict about type compatibility. On the other hand, the Ada definitions appear to be well constructed (although the author can not be certain of this as he is not an Ada programmer).

Apart from type definition mis-matches, the other major obstacle is the construction of string parameters. In the C programming language, a character string is an array of bytes terminated by a NULL byte (value 0). There is no such thing as a string descriptor; the length of the string is found by counting the characters up to the trailing NULL. C programs commonly pass strings between routines by using pointers to string variables. Thus, most string parameters for X11 routines are NULL-terminated arrays of bytes and this is how the MIT C binding declares them.

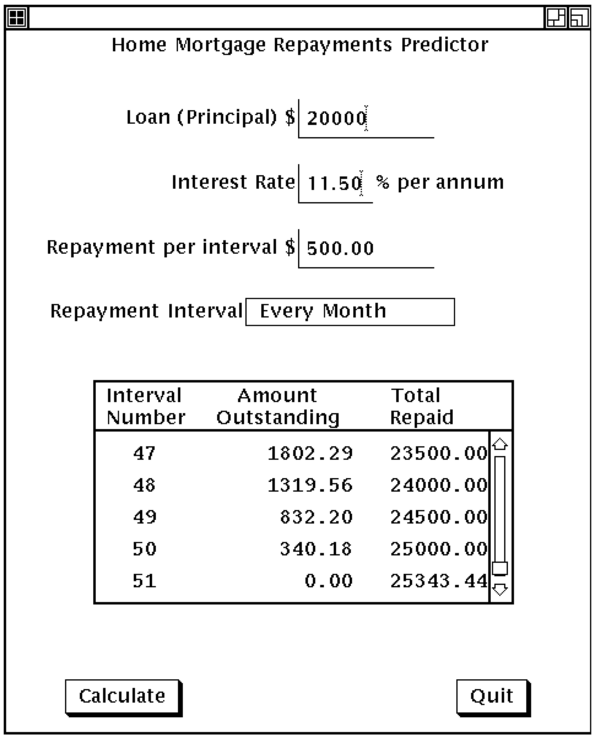
In the VMS binding, the DECwindows developers have taken steps to enable standard VMS-style strings to be used, with strings passed by descriptor. However, they did not want to completely rewrite the X11 software, so many DECwindows routines still deal with NULL-terminated strings. In general, a routine which has a string parameter will accept a VMS string descriptor (when using the VMS binding) but parameters which consist of arrays of strings, or which reference arrays of strings, are most likely to require a NULL-terminated array of bytes passed by reference.

A Real Test of DECwindows Without C

The inspiration for writing a DECwindows program in Pascal came from a Hypercard program running on a Macintosh which calculated home loan mortgage repayments (very topical at the time). It was thought to be not too hard to do, especially with the minimal set of inputs and outputs required, so attempts were begun. Pascal was chosed because it is the Author's favorite language and C is definitely not.

The Loan Program

The rest of this paper describes the Loan program, which is simple enough to understand but complex enough to demonstrate important features of DECwindows programming, including UIL interfacing. The pro-



gram's display is shown above, with an outline of the program at right.

Contrary to fundamentalist block structured programming discipline, the Loan program has a label. (The `Label` statement in Pascal declares a program label which will be defined later in the program; c.f. line numbers in BASIC or label numbers in FORTRAN.) A program label provides an easy way of aborting the program when a DECwindows call fails (or any other system service call, for that matter); statements after the label can be used to clean up and exit gracefully. An alternative is to call SYS$EXIT and use an exit handler, but exit handlers can be tricky when dealing with language RTLs (eg what state are files in, especially those opened by the language's RTL?).

Most of the rest of this paper describes the implementation of the Loan program. Each explanation refers to callouts agains a nominated Figure, which will be a section of program code.

Initialising the X Environment

In any DECwindows application which uses UIL there are two basic calls to set up the X11 environment. They are **Dwt$Initialize_DRM** and **Xt$Initialize**. Please refer to Figure 2.

1. A label isdeclared, as described earlier.
2. This is set up as a convenience for calling **Xt$Initialize** (see below). A trailing NULL is not re-

```
[inherit('SYS$LIBRARY:DECW$DWTDEF')]
Program LOAN (input,output);

Label
Exit;

Const
     ...
Type
     ...
Var
     ...
{       Utility Routines        }
{
{       Widget Callbacks        }

Begin {MAIN}
{ Initialize local data structures }

{ Initialize UIL/DRM }
Dwt$Initialize_DRM;

{ Initialize the Toolkit }
topLevel := Xt$Initialize(
                          progname
                         ,progname
                         ,0,0
                         ,argcount,);

{ Open the UIL/DRM Hierarchy }
ret_status := Dwt$Open_Hierarchy(
                      UIL_file_count
                     ,filenames
                     ,%immed 0
                     ,hierarchy);

{ Register names with DRM }
ret_status := Dwt$Register_DRM_Names (
                     callback_names
                    ,DRM_name_count);

{ Fetch and manage the widget tree }
ret_status := Dwt$Fetch_Widget(
                          hierarchy
                         ,'MAIN'
                         ,topLevel
                         ,main
                         ,class);

Xt$Manage_Child (main);

Xt$Realize_Widget (topLevel);

Xt$Main_Loop;

Exit:
end.
```

quired as the VMS binding for this routine will accept a string descriptor.

3. When using a UIL file, it is common to use symbolic names instead of magic numbers to identify such entities as widgets, gadgets or miscellaneous numeric values. In the UIL file for this program, the following declarations are made:

```
k_field_principal: 0;
k_field_rate:      1;
k_field_payment:   2;
```

   There is more on these definitions a bit later. Note the use of the `[long]` attribute on type `field_type` which tells the compiler that the enumerated type should be represented as a set of longword integer values. This is a precautionary step as all UIL integers are longword integers.

4. This program has three Text widgets, used for getting numeric input from the user. It is common DECwindows programming practice to use arrays of like widgets which makes callback processing simpler; the callback identifies the widget by array index which allows the program to associate other information with the widget if necessary. In this case the widget's name is held by the program along with the widget ID. If this were not done, and the program needed to know the widget's name, it would be necessary to use a call to **Xt$Get_Values** to determione the widget's name. The program statements at point **4** initialise the `input_fields` array; an alternative, in Pascal, would be to use a static initialisation or VALUE block.

5. When using DRM (DECwindows Resource Manager) the first task is to initialise it. This call is exactly the same as in the MIT C binding, except for the form of the routine name..

6. After setting up the DRM it is necessary to initialise the X Toolkit. This routine has already been briefy described for the "Hello World" program. The parameters for this routine are

FIGURE 2: Initialising the X Environment

```
1  Label    Exit;

2  Const    progname             = 'Loan';

3  Type     field_type           = [long] (principal, rate, payment);

   Var      topLevel, main       : Dwt$Widget;
            argcount             : Dwt$Cardinal := 0;
            input_fields         : array [principal..payment] of record
                                     widget : Dwt$Widget;
                                     name   : varying [30] of char;
4                                  end;

   Begin
   { Initialize local data structures }
   input_fields[principal].name := 'Loan';
   input_fields[rate].name      := 'Interest Rate';
   input_fields[payment].name   := 'Repayment';

5  { Initialize UIL/DRM }
   Dwt$Initialize_DRM;

   { Initialize the Toolkit }
   topLevel := Xt$Initialize (progname, progname, 0, 0, argcount,);
6  if topLevel = nil then
     begin
     writeln('Error creating topLevel');
     goto Exit
     end;


   Exit:
   end.
```

P1 - Application Name (program name)
P2 - Class Name
P3, P4, P5, P6 - Command line parsing. The command line parsing parameters are required in all programs, but only C programs will be likely to make full use of them. Note the declaration of the "argcount" parameter which DECwindows requires to be a variable.

The Class Name is used by the Xt (X Toolkit) routines to locate the resource file for the application. Resources for the application named in parameter 1 are prefixed by the application name in the resource file. In theory, one would have a single resource file called, for example, EDITORS.DAT, for all text editor programs; those programs would pass EDITORS as the Class Name and the individual program name as the Application Name. In practice, the application name and the class name tend to be the same, and each application has its own resource file. The resource file for this program is quite simple:

```
Loan.MAIN.child_overlap: False
Loan.MAIN.width: 350
Loan.MAIN.height: 240
```

More on UIL file constants

Quite often, constants are declared in the UIL file which must also be used in the program. Unfortunately there is no guaranteed way of making sure that the two sets of constants match. For example, it would be nice if the syntax of the UIL declaration and the syntax of the programming language used were the same, because this would allow the constants to be placed in a separate file for inclusion into the UIL file and the program. It could be done, by using some sort of preprocessor, but that would require a fair bit of extra work to set up and use. A better solution would be to put the declarations in a data dictionary, .e.g. Digital's CDA product, but that is not supported by UIL at this time. The usual way is to declare constants in the UIL file and in the program, and make sure that they match.

It would be possible to declare constants in the Pascal program, but a Pascal enumerated type provides an alternative mechanism. The use of an enumerated type simplifies debugging a Pascal program because the Pascal RTL converts enumerated values to their names wnen writing them to a text file (eg SYS$OUTPUT). One thing to be aware of is the base value of the enumerated type; Pascal sets the base value to 0, but some languages might not.

Opening the UIL file

This is where the problems with the VMS Binding really begin. Please refer to Figure 3.

The call to **Dwt$Open_Hierarchy** looks quite simple, but it took some time to work out the details of how to set up its parameters. This routine can open as many UIL files as desired by the programmer, for example to cater for different language variants of the application. Each file is indicated by an array element which points to a string descriptor for the file name. This is different from the MIT C binding where each array element points to the string itself —more on this in a moment.

In the degenerate case of one file it is necessary to declare a single-element array, as done here. It is not sufficient to pass the filename by descriptor alone because **Dwt$Open_Hierarchy** is expecting the address of a pointer to a descriptor. It would be quite acceptable to pass such a pointer, but this is a demonstration program so an array is used.

The third parameter is operating system dependent; in the VMS environment it can be used to supply a list of default file specifications for the UIL files listed in the first parameter.

The final parameter is a context variable which is used in later calls to UIL related routines.

When setting up the filenames list for **Dwt$Open_Hierarchy** the programmer must supply a string descriptor for each filename. In this program it is done by declaring a string descriptor type and declaring an array of pointers to elements of that type. The address of the filename can be found in Pascal using the **iaddress** function, but the mechanism of calling it must be done carefully. One possible (and obvious) method is the Pascal statement

```
iaddress('loan_display.uid')
```

This statement should return the address of the string `'loan_display.uid'` but instead returns the address of a temporary copy of that string, which is no use because the copy is likely be destroyed before **Dwt$Open_Hierarchy** is called. Declaring a constant whose value is the filename will not work because **iaddress** does not accept a constant for a parameter. So the filename must be set up as a variable, with a compile-time initialisation clause, which does the job.

Note that the filename has a trailing NULL. The routine definition calls for an array of pointers to string descriptors, but it was found that this is not sufficient: each string must be NULL-terminated as well. Omitting the NULL may work, if the compiler happens to allocate a NULL-valued byte following the string, but it is safer to specify it explicitly.

```
Const
     UIL_file_count         = 1;
Type
     strdescr               = record
                                    w_len     : [word] 0..65535;
                                    b_dtype   : [byte] 0..255;
                                    b_class   : [byte] 0..255;
                                    a_pointer : [unsafe] unsigned;
                                  end;
Var
     hierarchy              : Dwt$DRM_Hierarchy;
     UIL_filename           : packed array [1..30] of char := 'loan_display.uid'(0);
     filenames              : array [1..UIL_file_count] of ^strdescr;
     ancilliary_filenames   : array [1..UIL_file_count] of Dwt$IDBOS_Oparam := Zero;


{ Open the UIL/DRM Hierarchy }
new(filenames[1]);
with filenames[1]^ do
  begin
  w_len      := length(UIL_filename);
  b_dtype    := 14;  { DSC$K_DTYPE_T }
  b_class    := 1;   { DSC$K_CLASS_S }
  a_pointer  := iaddress(UIL_filename);
  end;
ret_status := Dwt$Open_Hierarchy (UIL_file_count, filenames, %immed 0, hierarchy);
if ret_status <> DWT$C_DRM_SUCCESS then
  begin
  writeln('Error opening UIL files');
  goto Exit
  end;
```

FIGURE 3: Opening the UIL File

At this point the program is running with its basic X environment established (ie the DECwindows Toolkit is initialised and a connection is open to the X server), and the UIL file has been opened. There is still more to do before the application becomes full active.

Linking the UIL to the Program

After the UIL file has been opened the program must associate the entities declared therein with the corresponding program routines and data structures. The is done by calling **Dwt$Register_DRM_Names** which has two parameters: a list of name/address pairs, and a count of the number of items in the list. Please refer to Figure 4.

The DECwindows X Toolkit (VMS Binding) provides a routine for creating the `callback_names` list, **VMS$Set_Arg**; in the MIT C binding this is a macro. However, this routine is rather inefficient as it takes copies of its parameters to build the list, and these copies must be manually deleted by the programmer (by calling **VMS$Free_Argnames**) when the list is no longer required. This might be OK for lists whose contents have to built from elements which are known only at run-time, but seemed to the Author to be a bit clumsy for static information. It would be better if Pascal could do the necessary initialisation in a VALUE block (or other compile-time construct) but it can't, so a way was found to quickly build a list at run-time from static information:

1. The `register_name` routine build the argument list for **Dwt$Register_DRM_Names**. The construction of its formal parameter list is rather important, particularly the first parameter. In order to deter Pascal from generating code which takes a temporary copy of the **rname** parameter, the formal parameter definition must declare it as a **var** parameter. However, in this program `register_name` is called with a constant string (eg `'SET_INTERVAL'(0)`, note the trailing NULL) which is not a legal actual parameter for a formal **var** parameter. Thus the `[readonly]` attribute which tells the compiler that the parameter will not be modified. The result is that the address of the string, and not a copy of it, is passed to `register_names`.

```
Label
  Exit;

Const
  DRM_name_count        = 6;

Var
  ret_status            : Dwt$Cardinal;
  callback_names        : array [1..DRM_name_count] of Dwt$DRMreg_Arg;


  {------------------------------------------------------------------------------
    Store a name/value pair for DRM REGISTER NAMES
  }
  procedure register_name (var rname : [readonly]
                                       packed array [l..h:integer] of char;     1
                           rvalue : unsigned;
                           var reg : Dwt$DRMreg_Arg);
  var p : record case integer of
           1 : ( i : unsigned);           2
           2 : ( s : Dwt$Strng);
          end;
  begin
  p.i := iaddress(rname);
  reg.Dwt$A_DRMr_name := p.s;
  reg.Dwt$L_DRMr_value := rvalue;
  end;
                                                        3
{ Register names with DRM }
register_name ('SET_INTERVAL'(0), iaddress(Set_Interval), callback_names[1]);
register_name ('QUIT'(0), iaddress(Quit), callback_names[5]);
register_name ('CREATE_LIST_BOX'(0), iaddress(Create_List_Box), callback_names[6]);
ret_status := Dwt$Register_DRM_Names (callback_names, DRM_name_count);
if ret_status <> DWT$C_DRM_SUCCESS then
  begin
  writeln('Failed to register DRM names');
  goto Exit
  end;


Exit:
end.
```

FIGURE 4: Registering entity names with Xt$DRM_Register_Names

2. In `register_name` a variant record is required to convert the address of the `rname` parameter to the appropriate DECwindows datatype. **Dwt$Strng** (the spelling is correct) is declared as a **pointer to char**, but the **iaddress** function returns an integer value, so its data type must be converted. Type-casting cannot be used because Pascal disallows its use on function results, hence the variant record.
3. The second parameter to `register_names`, in the above usage, is the address of a callback routine referenced by the UIL file. The **VMS$Set_Arg** routine allows strings without trailing NULLs to be registered, but only because it takes a copy of its string parameter. The technique used here avoids this overhead.

This technique works well for "static" declarations but has not been used in a more complex application. An alternative method of building these kinds of argument lists, which uses type-casting instead of a variant record, is given later in this paper.

Creating the Widgets

The last step in setting up the program is to create the display elements (widgets and gadgets) defined in the UIL file. Consider the following code fragment:

```
Var
   ret_status      : Dwt$ardinal;
   topLevel, main  : Dwt$Widget;
   class           : Dwt$DRM_type;
   hierarchy       : Dwt$DRM_Hierarchy;


{ Fetch and manage the widget tree }
ret_status := Dwt$Fetch_Widget (
                   hierarchy
                   ,'MAIN'
                   ,topLevel
                   ,main
                   ,class);
if ret_status <> DWT$C_DRM_SUCCESS then
  begin
  writeln('Error fetching widgets');
  goto Exit
end;
```

Dwt$Fetch_Widget is one of the simpler routines to use, as there are no strange parameters, no type conflicts, and no data allocation problems. The only thing to be careful of here is the use of the case sensitivity clause (`names=case_[in]sensitive`) in the UIL file: **case_insensitve** means the all named entities in the UIL file are referenced by the program using uppercase; if not, the DECwindows Toolkit won't find them. When using **case_sensitive**, the program and the UIL file must use the same case to refer to a given entity.

Notes on Callback Routines

This program has a number of callback routines which implement the user interaction. Figure 5 illustrates some simple callback routines used within the Loan program.

1. User data for callback routines is typically an integer value, although more complex data can be associated with the widget itself when required. Callback routines normally have three parameters: the widget, an integer user data value, and the callback reason. In this program, constant integers are used; an enumerated type is used to define constants corresponding to similar definitions in the UIL file.
2. As each text widget is created, the UIL does a callback to the program so that the widget declaration may be associated with the widget instance. The `Create_Data_Field` routine simply copies the callback user data to the `input_fields` array so that the program knows the widget ID of the respective widget.
3. The Loan program allows the user to specify the repayment interval by means of a pop-up option menu. Each menu entry is declared in the UIL file with a callback pointing to this routine, and the callback parameter indicates which entry is being created. There is no need for the program to know the widget IDs of the menu itself or even of each menu entry, because the program does not manipulate them directly. It just has to know which one is active.
4. Sometimes it is useful to know what the program is doing; this callback does a tracewrite to show what interval has been selected by the user. Note how the declaration of `interval_type` as an enumerated type helps here — the Pascal RTL converts it to a string automatically, instead of the program having to do that manually.
5. The List Box callback has no user data, as there is only one list box in the program (so no array of widget IDs is used). The program still needs to know the list box's widget ID to allow values to be written to the list box. It would have been possible to place all the widgets in one widget array but there is no relationship between the input fields and the list box, so a separate declaration is used.
6. Some callbacks have no parameters at all. When the Quit button is pressed, the program has to exit. The Exit label is very useful here, although a more complex program would probably use a different method to signal the end of processing.

The callback routines do not need to be declared with the `[global]` attribute (which tells the Pascal compiler to make global symbols for the entry points) because the VMS Linker does not have to link these routines to other object modules. The **Dwt$Register_DRM_Names** does this linkage at runtime.

On the other hand, the callbacks should probably be declared `[asynchronous]` because they are called outside of program control. Doing this would cause other problems, in that all of the variables which the callback routines access would have to be declared `[volatile]` or else the compiler would complain. In the Loan program (and probably most other DECwindows programs) the non-asynchronous usage is probably OK, because the routines themselves do not interrupt each other.

The UIL declarations for the above callback routines can be found in the program listing at the end of this paper.

```
1 Type
    interval_type         = [long] (weekly, fortnightly, monthly, bimonthly
                                   ,quarterly, half_yearly, yearly);
    field_type            = [long] (principal, rate, payment);

  Var
    input_fields          : array [principal..payment] of record
                              widget : Dwt$Widget;
                              name   : varying [30] of char;
                            end;
    list_box              : Dwt$Widget;
    selected_interval     : interval_type := monthly;


3   { Repayment Interval menu }
    Procedure Set_Interval (widget: Dwt$Widget; interval: interval_type;
                            reason: Dwt$Any_CB_St);
    begin
4   writeln ('in Set_Interval; selected interval = ', interval);
    selected_interval := interval;
    end;

6   { Quit button }
    Procedure Quit;
    begin
    writeln('in Quit');
    goto Exit;
    end;

5   { List Box }
    Procedure Create_List_Box (widget: Dwt$Widget);
    begin
    writeln('in Create_List_Box');
    list_box := widget;
    end;

2   { Text Input Fields }
    Procedure Create_Data_Field (widget: Dwt$Widget; f: field_type;
                                 reason: Dwt$Any_CB_St);
    begin
    writeln('in Create_Data_Field');
    input_fields[f].widget := widget;
    end;
```

FIGURE 5: Simple Callback Routines

More on Argument Lists

Many DECwindows Toolkit routines use argument lists to pass parameters between routines. The VMS binding provides several routines to build argument lists, all of which work by taking temporary copies of their paramters, and require that the temporary copies be destroyed by a call to **VMS$Free_Argnames**. Figure 6 shows a variation on the `register_names` procedure described earlier, to build an argument list containing name/value pairs which will be used to alter the a widget.

In this section of code, the **string_address** routine simply returns the address of its string parameter (this cannot be done directly using **iaddress**, because it can't be applied to constant strings). Instead of using a variant record to convert the integer value to a **Dwt$Strng** value, type casting is used. The net action of this code fragment is to empty a list box widget, which is dones by setting the `items` array to `0` and setting the `itemsCount` to `0`.

Ideally the program should use the predeclared constants (show in comments next to the code) which identify these widget resources, but the predeclared values do not include the trailing NULL required by the **Xt$Set_Values** routine. If the program used the supplied **VMS$Set_Arg** routines these predeclared names could be used.

Processing Simple Text Strings

Most DECwindows applications will have to display text at some point. This is very simple in Pascal when using simple strings (instead of the less frequently used compound strings).

In Figure 7, the **writev** built-in procedure is used to convert numbers to ASCII strings (there are many ways to do this using RTL routines or the $FAO system service), then **Dwt$S_Text_Set_String** is used to set the text value of the widget.

Using a Work Procedure

In the Loan program, an iterative procedure is used to calculate the loan repayments from the input data. It would be possible to have the program do this in a loop and present the final results when the calculation has finished, but a more graphical output was desired. In this program a list box displays intermediate results during the repayment calculation. It is generally a good idea when writing a DECwindows application to use a work procedure for any operation which is iterative in nature and has the potential for taking more than a moment of elapsed time to complete. Using a work procedure allows X events to be processed while calculating successive interest payments. In the Loan program, the work procedure is set up when the **Calculate** button is pressed; its job is to perform one iteration of the loan calculation.

FIGURE 6: Another Way to Build an Argument List

```
function string_address (var str : [readonly] packed array [l..h:integer] of char)
                         : integer;
begin
string_address := iaddress(str);
end;

{ Calculate Button }
Procedure Calculate_Button_Pressed;
   ...
{ Clear out the old entries in the list box }
if interval_number > 0 then
begin
i := string_address('items'(0));                 { Dwt$C_Nitems }
args[1].dwt$a_arg_name := i::Dwt$Strng;
args[1].dwt$l_arg_value := 0;
i := string_address('itemsCount'(0));            { Dwt$C_NitemsCount }
args[2].dwt$a_arg_name := i::Dwt$Strng;
args[2].dwt$l_arg_value := 0;
Xt$Set_Values (list_box, args, 2);
interval_number := 0;
end;
```

FIGURE 7: Processing Simple Text Strings

```
var s : varying [10] of char;

{ Update the input fields to reflect the values in use }
writev(s,initial_loan:1);
Dwt$S_Text_Set_String(input_fields[principal].widget, s);
writev(s,interest_rate:1:2);
Dwt$S_Text_Set_String(input_fields[rate].widget, s);
writev(s,repayment_amount:1:2);
Dwt$S_Text_Set_String(input_fields[payment].widget, s);
```

`Calculate_One_Interval` is a function in the Loan program with no parameters and which returns a boolean result. The code below prepares the two parameters for the DECwindows routine **Xt$Add_Work_Proc**. In this example the two parameters have the same value (for want of a better thing to do!) but this would not normally be the case.

```
i := iaddress(Calculate_One_Interval);
work_proc := i::Dwt$Work_Proc;
client_data := i::Dwt$Opaque;
Xt$Add_Work_Proc (work_proc
                 ,client_data);
```

Setting up the work procedure would have been much easier if the **Xt$Add_Work_Proc** routine had a formal parameter list which allowed the work procedure to be passed directly, as is done for AST routines when calling VMS system services such as $QIO.

The work procedure is called by the DECwindows Toolkit whenever there are no X events to process, and continues to do so until the work procedure returns **`false`**.

The work procedure performs the calculations for one repayment interval, then updates the list box to show the latest results. A list box displays compound strings which are generated quite easily using the VMS binding. as shown in Figure 8.

1. The Pascal function **writev** produces the ASCII string to display.
2. The ASCII string is converted to a Latin1 comound string (the most commonly used compound string).
3. The compound string is placed at the end of the list. The value `0` means "end of list"; `1` means the head of the list, and any other value refers to a particular position in the list.
4. The display is updated using **Dwt$List_Box_Set_Pos** to bring the new list entry into view.
5. The list box has its own copy of the string, so the copy created by **Dwt$Latin1_String** is destroyed using **Xt$Free**.
6. The work procedure indicates whether it should be reinvoked by returning a **`true`** or **`false`** value.

Conclusion

Many routines and structures can be used and declared in the same way in both the VMS binding and MIT C binding; if this were not the case, there would be little point in trying to write DECwindows programs in any language other than C.

The VMS binding generall handles string parameters quite well, but is not so good when a routine has a list (ie an array) of strings as a parameter. The VMS binding is just that — an interface to the X11 Window System which does some parameter type conversion and little else. The supplied convenience routines **VMS$Set_Callback_Arg**, **VMS$Set_Arg** and **VMS$Set_Desc_Arg** work OK but suffer from two deficiencies compared to the MIT C binding (**XtSetArg** macro):

- There is a different routine for each type of argument list, whereas **XtSetArg** handles different argument types better.

FIGURE 8: A Simple Work Procedure

```
  Function Calculate_One_Interval {(var loan_status : loan_rec)} : boolean;
  var s                  : varying [40] of char;
      cs                 : Dwt$Comp_String;
      actual_repayment   : double;
      new_value          : double;
      insufficient       : boolean;

  begin
  { Make the repayment }
        .
        .
  { Display the result }
1 writev(s,interval_number:3,current_value:16:2,total_paid:11:2);
2 Dwt$Latin1_String(s,cs);
3 Dwt$List_Box_Add_Item(list_box, cs, 0);
4 Dwt$List_Box_Set_Pos(list_box, interval_number);
5 Xt$Free (cs);
6 Calculate_One_Interval := (current_value <= 0.0) or insufficient;
  end;
```

- The program must call **VMS$Free_Argnames** after the argument list has been used, otherwise it will eventually run out of memory.

DECwindows programs, like all X11 programs, are asynchronous in nature and so it is possible that compiler optimizations might cause strange bugs to appear if appropriate steps are not taken. For example, declaring routines to be `[asynchronous]` and variables to be `[volatile]`, as appropriate. However, the author has not found this to be necessary in the few simple programs written so far.

The **biggest** problem with using the VMS binding concerns the declaration of formal parameters for DECwindows routines. Some of these are particularly poor, eg **Xt$Add_Work_Proc**, but no doubt this is due in many cases to the nature of SDL (the means by which the language-specific definition files are produced).

On the other hand, C is relatively lax about type checking and most DECwindows routine definitions in the `#include` files do not even define the formal parameters. This makes calling the routine much easier to code, but also quite possbly leads to more bugs. Things may be different in FORTRAN or BASIC which traditionally apply less stringent checks to formal and actual parameters.

In summary: yes, it can be done. If you want to write DECwindows programs and you don't like C, or don't have a C compiler, you can still do the job. Even in VAX TPU!