t Writing XPrims "557"

This section uses four example XPrims to illustrate how to write and build your own XPrims.

XPrims, if written properly, will run in both interpreted and compiled applications without alteration. The interpreter specific code should be bracketed by the `#if _INTERPRETER` and `#endif` statements. This code is then included only if the `_INTERPRETER` flag is set to one. To use the XPrims as part of a compiled application, you need only set the flag to zero.

Code which is interpreter specific and should be bracketed includes:

q A Main routine (described below) which loads your primitives into the Prograph interpreter's primitive table, initializes the global data register, and calls the AddPrimitive supplied function for each XPrim to be added;

q
calls to the CallPrimitive function;

q code to check for data related errors.

XPrim names must follow the naming conventions for universal methods. This is because the compiler uses the same conventions for primitives and for universal methods. The compiler automatically identifies all C functions which begin with "U_" as universal methods.

The name of each XPrim's C function must be of the form U_name. Any non-alphanumeric characters in the name must be replaced by their two-character, hexadecimal ASCII code and delimited by underscores. For example, the primitive point-in-rect? is represented as U_point_2D_in_2D_rect_3F_ (2D and 3F being the ASCII representations of '-' and '?' respectively).

There are several differences between interpreted and compiled code which should be kept in mind when writing XPrims:

q Prograph primitives can be called directly in compiled code. You must, however, set the arity with the SETARITY macro (described above in "Arity Macros") before doing so. For example, the code to call the show primitive with three inputs would be:

```
#if _INTERPRETER
 CallPrimitive( "\Pshow", 0x0300, myString1, myInteger,
              myString2 );
#else
 SETARITY( 3, 0 );
 U_show( myString1, myInteger, myString2 );
#endif
```

q The supplied functions AddPrimitive and CallPrimitive are not used in compiled code.

q The index parameter of list functions (ListEmptySlot, for instance) is four bytes in compiled code and two bytes in interpreted code.

q The size parameter of list and string functions (StrStretchString, for instance) is four bytes in compiled code and two bytes in interpreted code

q The slot parameter of MakeC_List is four bytes in compiled code and two bytes in interpreted code

Function Return Values "559"

When a primtive completes without error, its C function should return PCF_TRUE for its integer return parameter.   The only exception is a boolean primtive with no output root; such a method, if it completes successfully, conveys its boolean result by passing either PCF_TRUE or PCF_FALSE.

NOTE: In the flags passed to AddPrimitive, PF_CTRL should be specified for a primitive that can return either PCF_TRUE or PCF_FALSE; this forces a default next-case control on the calling operation. PF_VAR should be specified if the primitive has an optional boolean output.

Error conditions are reported to the interpreter by returning one of the following error codes:

```
/* primitive execution errors */

#define PRIMERR_ARITY
 0x0100 /* wrong # of inputs/outputs */
#define PRIMERR_TYPE
  0x0200 /* wrong type of input */
#define PRIMERR_ARITH
 0x0300 /* arithmetic error */
#define PRIMERR_VALUE
 0x0400 /* input value out of range */
#define PRIMERR_COMPARE
 0x0500 /* incomparable types */
```

These error codes are recognized by the Prograph interpreter only. In compiled code PCF_TRUE and PCF_FALSE are the only valid return codes.

In interpreted code the following error checks should be made:

All primitives, whether of fixed or variable arity, should verify their arity and return PRIMERR_ARITY if the arity is not correct.

The types of all inputs should be validated. If an input type is incorrect, PRIMERR_TYPE should be returned.

If the range of an input value is not acceptable, PRIMERR_VALUE should be returned.
For PRIMERR_TYPE and PRIMERR_VALUE errors, the ordinal number of the input containing the bad data should be added to the error value. For example, if the second input of a primitive should have been an integer but was a boolean instead, the value returned should be PRIMERR_TYPE+2.

In compiled code values such as PRIMERR_ARITY, PRIMERR_VALUE and PRIMERR_TYPE should never be returned by methods. Error checking should be done when running in the interpreter but not in compiled code.   A C compiler directive flag, `_INTERPRETER` can be used to bracket error checking code.

The XPrim examples illustrate this technique.

The next two sections describe how to build a set of example XPrims as compiled and interpreted code respectively. Instructions are given for THINK C and MPW C. The example XPrims themselves are described in a separate section.

XPrims in Interpreted Code *560*

The Prograph interpreter does not recognize XPrims in the same way as the compiler. For each XPrim, you must supply a 'STR#' resource containing the primitive's name and help information. This information is used to identify primitives and is also displayed in Prograph's ˙Info… dialog. You must also have a main routine which calls AddPrimitive for each primitive.

Each 'STR#' must consist of three strings: the mnemonic names of the input and output arguments, the types of the input and output arguments, and a brief description of what the primitive does. Three strings must be present in each 'STR#' resource, even if they are empty.

The 'STR#'s resource number is used to map the code for a primitive onto its associated 'STR#'. The 'STR#' resource number can be any number between 1 and 32K but must be unique within its file.

The 'STR#'s resource name becomes the name of the Prograph primitive. Only the first 19 characters of a primitive's name are recognized uniquely.

Each 'STR#' resource should be marked as purgeable.

The value of a global data register (A4 in THINK C, A5 in MPW C) is set up at the beginning of the main routine. The data register is automatically restored by the interpreter before an XPrim is called. This allows the code resource to have global or static data.

In the main routine, a call is made to AddPrimitive for each of your primitives as shown in the following example:

```
 AddPrimitive( 4, 0x0001, PF_USER, 0, &U_get_2D_filter );
```

The first argument to AddPrimitive is the resource number of the primitive's associated 'STR#' resource.

The second argument to AddPrimitive describes the default arity given to the primitive's calling operation when created in a Prograph program. Note that this is not necessarily the same arity that is passed to the call to the primitive.

The third argument is a set of flags. All XPrims need to have the value PF_USER set as one of the flags. A fixed-arity primitive, which is created without a control, should have a flags argument of PF_USER. If the primitive has variable-arity, the flag PF_VAR should be set. If the calling operation is to be created with a default next-case control, the flag PF_CTRL should be set. These flags are defined in X_constants.h.
The fourth argument should always be zero.

The last argument is the address of the primitive's code.

Building Interpreted XPrims Using THINK C *561*

This section describes how to build and use the example XPrims using THINK C.

u
Create a new THINK C project.

u
Select Set Project from the Project menu, set the project type to “Code Resource,” set Type to 'TGSC', and set ID to 1.

u
Set File Type to 'TGSC' and Creator to TGSL.

u
Add to the project the version of MacTraps supplied with THINK C, and the files XP_support_I and example_XPrims.c supplied with Prograph.

u Make sure the `_INTERPRETER` flag in X_includes.h is set to 1.

u
Compile and build the code resource into a file called example_XPrims.

u Move example_XPrims into the same folder as Prograph and the example XPrims are ready to use.

NOTE: 'TGSC' resources can be numbered arbitrarily. The maximum size of a 'TGSC' code resource is 32K bytes. However, if your code exceeds 32K, you can use the mutli-segment code resource option in THINK C.

Creating the 'STR#' Resources Using THINK C*561*

Each primitive should have a corresponding 'STR#' resource containing the primitive’s name and help information; this will be displayed in Prograph’s ˙Info… dialog.

In THINK C when the resource file name is the project name suffixed with '.rsrc,' the 'STR#' resources are automatically copied over whenever the code resource is built. You can copy an 'STR#' resource from the example_XPrims_I.π.rsrc file and use it as a template for your own primitives.

The file example_XPrims_I.π.r contains the 'STR#' resources for the example XPrims. You can use this file as a template for creating your own 'STR#' resources. Run this file through RMaker , and then use ResEdit to copy the resources into the file created when your code resource is built.

Here are the contents of example_XPrims_I.π.r :

```
example_XPrims_I.π.rsrc

TYPE STR#

list-average, 1 (32)
3
Inputs:   TheList\0D++
Outputs:   TheAverage
```

```
Inputs:   list\0D++
Outputs:   real
Accept a list of numbers, return the average.

input-average, 2 (32)
3
Inputs:  TheNumber; [ TheNumber; …]\0D++
Outputs:  TheAverage
Inputs:   number; [number; …]\0D++
Outputs:   real
Accept 1 to n numeric inputs, return the average.

point-in-rect?, 3 (32)
3
Inputs:   ThePoint; TheRect\0D++
Outputs:   [TheResult]
Inputs:   point; rect\0D++
Outputs:   [boolean]
ThePoint is in TheRect? With output the result is TRUE or ++
FALSE. Without output the primitive succeeds or fails.

get-filter, 4 (32)
3
Inputs:   \0D++
Outputs:   TheFunctionPointer
Inputs:   \0D++
Outputs:   ABlock@
Return TheFunctionPointer. ( Used for FileFilter argument in call to
SFGetFile.)
```

The Main Routine in THINK C *563*

Each 'TGSC' code resource contains one main routine. This routine initializes Prograph primitive information at program startup.   The main routine is called once, during Prograph initialization. The code for the THINK C main routine from example_XPrims.c is as follows.

```
#if _INTERPRETER

/* main loads primitives into interpreter's XPrim table */

void main( table )

ProcPtr * table;
  /* common table */

 asm

  move.l a4, -(sp)
/* move A4 on to the stack */
  move.l a0, a4
  /* move address of code resource to A4 */


 ct = table;
 /* set up common table for shared functions */

 AddPrimitive( 1, 0x0101, PF_USER, 0, &U_list_2D_average );
 AddPrimitive( 2, 0x0101, PF_USER | PF_VAR, 0,
               &U_input_2D_average );
 AddPrimitive( 3, 0x0200, PF_USER | PF_CTRL | PF_VAR, 0,
               &U_point_2D_in_2D_rect_3F_ );
 AddPrimitive( 4, 0x0001, PF_USER, 0, &U_get_2D_filter );

 asm  move.l (sp)+, a4
 /* move saved value back into A4 */


#endif
```

The value of register A4 is set up at the beginning of the main routine. The current value of A4 is recorded, along with the primitive information, during a primitive's initialization. At the end of the main routine, the previous value of A4 is restored. Whenever a primitive is called, its saved A4 value is automatically restored. This allows the primitive to access any of the code resource's global or static data.

In the main routine, a call is made to AddPrimitive for each primitive in the 'TGSC' resource.

Building Interpreted XPrims Using MPW C "564"

This section describes how to build and use the example XPrims using MPW C.

u
Copy the folders XPExamples, XPIncludes and XPLibraries into your MPW Examples, Interfaces and Libraries folders respectively.

u
Insert the following lines into your MPW startup script and execute them:

```
  Set XPIncludes "MPWInterfaces:  XPIncludes:  "
   Export XPIncludes


  Set XPLibraries "MPWLibraries:  XPLibraries:  "
   Export XPLibraries
```

u
Set the current directory to "MPWExample: XPExamples" and execute the command:

```
   make -f example_XPrims_I.make
```

u
Compile and build the code resource by executing the commands produced by the make command. The code resource file, example_XPrims, will be built with a File Type of 'TGSC' and Creator of TGSL.

u Move example_XPrims into the same folder as Prograph and the example XPrims are ready to use.

NOTE: 'TGSC' resources can be numbered arbitrarily. The maximum size of a 'TGSC' code resource is 32K bytes. However, if your code exceeds 32K, you can set the MPW linker flag for code resources greater than 32K.

Creating the 'STR#' Resources with MPW *564*

Each primitive should have a corresponding 'STR#' resource containing the primitive's name and help information; this will be displayed in Prograph's ˙Info… dialog.

In MPW, Rez can be used to append the 'STR#' resource to your primitive resource file. The file example_XPrims_I.r contains Rez source code for the 'STR#' resources for the example primitives. You can use this file as a template for creating your own 'STR#' resources. Here are the contents of example_XPrims_I.r:

```
#include "Types.r"

resource 'STR#' (1, "list-average", purgeable)

/* array StringArray:   3 elements */
  /* [1] */
  "Inputs:   TheList\n"
  "Outputs:   TheAverage",
  /* [2] */
  "Inputs:   list\n"
  "Outputs:   real",
  /* [3] */
  "Accept a list of numbers, return the average."

;

resource 'STR#' (2, "input-average", purgeable)

/* array StringArray:   3 elements */
  /* [1] */
  "Inputs:  TheNumber; [ TheNumber; …]\n"
  "Outputs:  TheAverage",
  /* [2] */
  "Inputs:   number; [number; …]\n"
  "Outputs:   real",
  /* [3] */
  "Accept 1 to n numeric inputs, return the average."

;

resource 'STR#' (3, "point-in-rect?", purgeable)

/* array StringArray:   3 elements */
  /* [1] */
  "Inputs:   ThePoint; TheRect\n"
  "Outputs:   [TheResult]",
  /* [2] */
  "Inputs:   point; rect\n"
  "Outputs:   [boolean]",
  /* [3] */
  "ThePoint is in TheRect? With output the "
  "result is TRUE or FALSE. Without output "
  "the primitive succeeds or fails."

;

resource 'STR#' (4, "get-filter", purgeable)

/* array StringArray:   3 elements */
  /* [1] */
  "Inputs:   \n"
  "Outputs:   TheFunctionPointer",
  /* [2] */
  "Inputs:   \n"
  "Outputs:   ABlock@",
  /* [3] */
  "Return TheFunctionPointer. ( Used for "
  "FileFilter argument in call to SFGetFile.)"

;
```

The Main Routine in MPW C "566"

Each 'TGSC' code resource contains one main routine. This routine initializes Prograph primitive information at program startup.   The main routine is called once, during Prograph initialization. The code for the MPW C main routine from example_XPrims.c is as follows.

```
#if _INTERPRETER

ProcPtr
* ct;
Int4
 PrographA5;

/* main - loads primitives into interpreter's XPrim table
 *      - creates area for global data
 */

void main( table )
ProcPtr * table;
   /* common table */

Int4
 XPrimA5;
Handle
 XPrimA5Handle;

 /* Create a global data area */

 XPrimA5Handle = NewHandle( (Nat4) A5Size());
 if ( XPrimA5Handle == NULL )
  return;

 MoveHHi(XPrimA5Handle);
 HLock( XPrimA5Handle );
 XPrimA5 = *XPrimA5Handle + (Int4) A5Size() - 32;
 A5Init( XPrimA5  );

 PrographA5 = SetA5( XPrimA5 ); /* Save value of A5 */

 ct = table;

 AddPrimitive( 1, 0x0101, PF_USER, 0,
         (ProcPtr) &U_list_2D_average );
 AddPrimitive( 2, 0x0101, PF_USER | PF_VAR, 0,
         (ProcPtr) &U_input_2D_average );
 AddPrimitive( 3, 0x0200, PF_USER | PF_CTRL | PF_VAR, 0,
         (ProcPtr) &U_point_2D_in_2D_rect_3F_ );
 AddPrimitive( 4, 0x0001, PF_USER, 0,
         (ProcPtr) &U_get_2D_filter );

 SetA5( PrographA5 );
/* Restore the value of A5 */


#endif
```

A global data area for the code resource is created at the beginning of the main routine and the current value of A5 is recorded. At the end of the main routine, the value of A5 is restored. The AddPrimitive routine records the A5 value of the code resource and whenever the primitive is called, this A5 value is automatically restored. If a primitive executes a supplied function which is a callback to the Prograph Interpreter the value of A5 is automatically saved and reset. This allows primitives to access the code resource's global or static data.

In the main routine, a call is made to AddPrimitive for each primitive in the 'TGSC' resource.

XPrims in Compiled Code *567*

The 'STR#' resource and the main routine required by the interpreter are not needed for compiled code. This is because the Prograph compiler first converts your C object code into it's own format, then links the converted object file directly with your compiled Prograph code.

Building Compiled XPrims Using THINK C *567*

To add the example primitives to a compiled application, perform the following steps:

u Create a new THINK C project.

u Add the file example_XPrims.c to the project.

u Make sure the `_INTERPRETER` flag in X_includes.h is set to 0.

u Compile and build a library and save it in a file called example_XPrims.lib.

u Include example_XPrims.lib in your Prograph project.

Building Compiled XPrims Using MPW C *567*

To add the example primitives to a compiled application, perform the following steps:

u
Copy the folders XPExamples, XPIncludes and XPLibraries into your MPW Examples, Interfaces and Libraries folders respectively.

u
Insert the following lines into your MPW start up script and execute them:

```
  Set XPIncludes "MPWInterfaces:  XPIncludes:  "
   Export XPIncludes


  Set XPLibraries "MPWLibraries:  XPLibraries:  "
   Export XPLibraries
```

u
Set the current directory to "MPWExample: XPExamples" and execute the command:

```
   make -f example_XPrims_C.make
```

u
Compile the example program by executing the command produced by the make command. This will create the object file example_XPrims_C.c.o.

u Include the object file example_XPrims_C.c.o and the library MPWLibrary in your Prograph project.

Example XPrims *568*

This section describes the four example XPrims. Note that the code is the same for MPW C and THINK C.

Example # 1

A fixed arity primitive, called list-average, with one input, consisting of the list of numbers to be averaged, and one output, the average of the numbers.

```
/* list-average - accept a C_list of numbers, return the average

              value of the list as a C_real */

Nat2 U_list_2D_average( input, output )
```

```
C_list *input;
  /* C_list of numbers */
C_real **output;
 /* address to put handle of output value */



Nat2
inarity;
  /* number of inputs */
Nat2
outarity;
 /* number of outputs */
Nat2
num_elements;
/* number of elements summed */
Nat2
index;
   /* current position in the list */
Handle item;
   /* generic handle to list item */
Real
total;
   /* total of items */

 GETARITY( inarity, outarity )

#if _INTERPRETER
 if ( inarity != 1 || outarity != 1 )
  return PRIMERR_ARITY;
    /* bad arity */

 if ( !IsType( input, C_LIST )  )
  return PRIMERR_TYPE + 1;
   /* wrong type on input 1 */
#endif

 num_elements = (**input).length;
 /* extract size for speed */

#if _INTERPRETER
 if ( num_elements == 0 )
  return PRIMERR_VALUE + 1;
   /* bad value on input 1 */
#endif

 for(index=0, total=0.0; index < (**input)length; index++)

  item = (**input).data[index];
/* get item from list */

  if ( IsType( item, C_INTEGER ) )
   total += ( **(C_integer *) item ).value;
  else if ( IsType( item, C_REAL ) )
   total += ( **(C_real *) item ).value;
#if _INTERPRETER
```

```
  else
   return PRIMERR_VALUE +1;
 /* type not numeric */
#endif


 *output = MakeC_real( total/num_elements );


 return PCF_TRUE;
     /* no errors encountered */
```

If inarity is not one and outarity is not one, the error code PRIMERR_ARITY is returned.

The IsType function verifies the type of input. If the item is not a C_list, PRIMERR_TYPE+1 is returned, indicating a type error on input 1.

The length of the list is extracted. If the size is zero, PRIMERR_VALUE+1 is returned, indicating a value error on input 1. (The size must be nonzero to avoid division by zero when calculating the average.)

Each item in the list is extracted and its type checked. If a particular item is neither an integer nor a real, a type error is returned for input 1; otherwise, the item's value is added to a running total.

The average is computed, passed to the MakeC_real function, and assigned to the output root.   The only "permanent" reference to the new C_real is on the output root, so the use count of one automatically provided by MakeC_real is correct.

The very last action of a primitive is usually to return the value PCF_TRUE   This indicates that the call to the primitive operation has completed without error, and basically instructs the interpreter or compiled code to proceed to the next operation.

Example # 2 *570*

A variable-arity primitive, called input-average, that returns the average value of its variable number of inputs.

```
/* input-average - accept 1 to n numeric inputs, return the
       average value of the inputs as a C_real */

Nat2 U_input_2D_average( args )

C_object *args;
   /* input & output args */


C_object *input;
  /* generic pointer to input args */
C_object **output;
  /* generic pointer to output arg places */
Nat2
 inarity;
```

```
  /* # inputs */
Nat2
 outarity;
 /* # outputs */
Int2
 index;
   /* current input */
C_object *item;
   /* generic handle to input  item */
Real10
total;
   /* total of inputs */
Nat2
 num_elements;
/* number of elements summed */

 VARITY( args, inarity, outarity, input, output );
              /* variable arity setup */

#if _INTERPRETER
 if( inarity < 1 || outarity != 1 )
  return PRIMERR_ARITY;
   /* bad minimum  arity */
#endif

 num_elements = inarity;

 for( index = 0, total = 0.0 ; index < inarity; index++ )

  item = input[index];
    /* get item from input */

  if ( IsType( item, C_INTEGER ) )
   total += ( **(C_integer *) item ).value;
  else if ( IsType( item, C_REAL ) )
   total += ( **(C_real *) item ).value;
#if _INTERPRETER
  else
   return PRIMERR_VALUE + index + 1;
/* type not numeric */
#endif


 *output[0] = MakeC_real( total/num_elements );

 return PCF_TRUE;
     /* no errors encountered */
```

This example shows how to create a variable-arity primitive. First the VARITY macro unpacks the primitive's inarity, outarity, inputs, and outputs. A check is then made on the range of the calling arity. This call must have at least one input and can have only one output; otherwise an arity error is returned.

The function iterates through the array of inputs, adding the input value to the total depending on the input's type. If the type of the input is neither C_integer nor C_real, a type error is returned together with the ordinal position of the invalid input.

In this example, the output is an array of addresses where output data can be placed.
In this case, there is only one output. The average is passed to the MakeC_real function and assigned to the output root.

The return value PCF_TRUE indicates that the primitive has completed without error.

Example # 3 *571*

A fixed-arity boolean primitive called point-in-rect?, which takes a point (or a pointer or handle to a point) and a rectangle (or a pointer or handle to a rectangle) and determines if the point falls within the rectangle. This primitive is initialized with a PF_CTRL flag set to indicate that it should be created with a next-case control.

If the call to the primitive has no output then it will either succeed or fail. If it has an output then the primitive will succeed and will return a C_boolean data item set to either TRUE or FALSE.

```
 /* point-in-rect? - boolean prim to determine if point is in a
          rectangle */

Nat2 U_point_2D_in_2D_rect_3F_( input1, input2, output )

C_Point
 *input1;
 /* input arg #1, the point  */
C_Rect
  *input2;
 /* input arg #2, the rectangle */
C_boolean
**output;
/* the result, if out arity = 1 */



Nat2
inarity;
   /* # inputs */
Nat2
outarity;
  /* # outputs */
Rect
r;
     /* value of input 1 */
Point
p;
     /* value of input 2 */
Bool
result;
   /* result of point in rect */

 GETARITY( inarity, outarity )

#if _INTERPRETER
```

```
 if ( inarity != 2 || outarity > 1 )
  return PRIMERR_ARITY;
   /* bad arity */

 if ( !HasType( input1, C_POINT ) )
  return PRIMERR_TYPE + 1;
  /* wrong type on input 1 */

 if ( !HasType( input2, C_RECT ) )
  return PRIMERR_TYPE + 2;
  /* wrong type on input 2 */
#endif

 switch ( GetRefLevel( input1 ) )

  case 2:
        /* handle to point*/
   p = **(Point **)(**(C_Handle *)input1).value;
   break;
  case 1:
        /* pointer to point*/
   p = *(Point *)(**(C_Ptr *)input1).value;
   break;
  default:
        /* point */
   p = (**input1).value;



 switch ( GetRefLevel( input2 ) )

  case 2:
        /* handle to rect */
   r = **(Rect **)(**(C_Handle *)input2).value;
   break;
  case 1:
        /* pointer to rect */
   r = *(Rect *)(**(C_Ptr *)input2).value;
   break;
  default:
        /* rect */
   r = (**input2).value;



 result = PtInRect( p, &r );

 if ( outarity == 1 )
  *output = MakeC_boolean( result );
 else if ( result == FALSE )

  return PCF_FALSE;

 return PCF_TRUE;
```

Since it is valid for this primitive to have either one or no outputs, the VARITY macro could have been used. VARITY was not used, partly for the sake of illustration, and partly because there is no need to set the inputs and outputs up as arrays.

First, the inarity and outarity are extracted and an arity range check is made. Checks are then made to ensure that input1 and input2 refer to a point and a rectangle, respectively. The supplied function HasType returns TRUE if input1's data item is a point, or a pointer or handle to a point; otherwise a type error is returned indicating which input is incorrect.

The function GetRefLevel switches to an appropriate case, based on the data item's level of indirection, and assigns the point to p and the rectangle to r.

The Toolbox routine PtInRect() is called. If an output root was specified, the result is packaged into a C_boolean and placed on the root, and PCF_TRUE is returned to indicate successful completion.

If the primitive does not have an output, the result of the PtInRect() call is returned as the function result: PCF_TRUE for TRUE, PCF_FALSE for FALSE.

Example # 4 *573*

Certain advanced features of the Macintosh interface are accomplished by passing function pointers to Toolbox calls. Such a custom function must be written in C in a code resource; the code resource should also contain an XPrim that returns the address of the custom function. From Prograph, you can then call the XPrim and pass the output to a Mac Method call.

The following example contains a filter function for the SFGetFile procedure and an XPrim, called get-filter, that returns the address of the filter function. The address of the filter function is then returned as the output of the primitive get-filter.

```
#if _INTERPRETER

/* display only those files whose names begin with e or E */

pascal Boolean FileFilter( block )

ParmBlkPtr block;


char c;

 c = block->fileParam.ioNamePtr[1];

 if ( c == 'e' || c == 'E' )

return FALSE;
    else

return TRUE;


/* get-filter - return pointer to FileFilter as an ABlock@ */
```

```
Nat2 U_get_2D_filter( output )


C_Ptr **output;
/* address to put handle of output value */


Nat2
inarity;
  /* # inputs */
Nat2
outarity;
 /* # outputs */

 GETARITY( inarity, outarity )

 if ( inarity != 0 || outarity != 1 )
  return PRIMERR_ARITY;
  /* bad arity */

    *output = MakeC_Ptr( NIL, &FileFilter );
 return PCF_TRUE;
    /* no errors encountered */


#endif
```

The function FileFilter, declared as using pascal calling conventions, returns a boolean value of FALSE if the file is to be included in the dialog’s list, or TRUE if it is not. The first character of the file’s name is examined.

The get-filter primitive first checks its arity. If the arity is correct, the address of the FileFilter function is passed to a new C_Ptr and assigned to the primitive’s output root. The function then returns PCF_TRUE to indicate completion without error.