Subject: linkimage tutorial
Posted by Joel Gales on Mon, 17 Dec 2001 16:26:52 GMT
View Forum Message <> Reply to Message

I've developed a short tutorial for creating linkimage modules in IDL.
I concentrate primarily on the IDL-C interface such as accessing and
creating IDL variable within the C module as this seems to be the
distinguishing aspect of such modules. I hope it proves useful and
removes some of the mystery of this powerful IDL feature. I welcome
your comments, criticisms, and corrections.

Joel Gales
SIMBIOS Code 970.2
Phone: (301) 286-1403
FAX: (301) 286-0268

++ +++

The purpose of this tutorial is to supplement the IDL External
Development Guide and help the reader better understand the interface
between IDL and C language linkimage modules.

Through a series of examples, we will show how (1) to access a
previously allocated IDL variable, (2) to create a new IDL variable, and

(3) to return these new variables to the IDL procedures or functions
from which they are called. These examples will be rather
simple-minded as we wish to concentrate on the interface. We will
start with scalar variables, proceed to array variables, and finish
with a discussion of string variables. The reader is expected to have
some understanding of the concepts of pointers in C, particularly,
pointers to structures. (Note: Chapter 9 of the External Development
Guide gives the detailed definition the of IDL_VARIABLE structure and
should be referred to by the reader through this tutorial.)

Linkimage C modules are compiled and linked in much the same way as
call_external
routines, however it is necessary to include the location of the
"export.h" header file (typically found in the idl "external" directory)
in the compilation step. Please consult the IDL documentation for your
platform.

Below is an IDL procedure that acts as a wrapper around the LINKIMAGE
statement. After compiling the C module and entering IDL, type:

IDL> li, 'quad1', 0

to linkimage a procedure or

IDL> li, 'quad3', 1

to linkimage a function.

```
pro li,shr_obj_name,code,dir=dir,entry=entry,silent=silent

IF (N_PARAMS() EQ 0) THEN BEGIN
        PRINT, 'li,shr_obj_name,code,<dir=dir>,<entry=entry>'
        PRINT, ' '
        PRINT, 'where "shr_obj_name" is the shared object filename.'
        PRINT, '      "code" is 0 for PRO, 1 for FUNCTION.'
        PRINT, '      "dir" is the directory of the shared object file.'

        PRINT, '      "entry" is the routine name.'
        PRINT, ' '
        PRINT, '      "entry" is needed if there is more that one
routine'
        PRINT, '      in the shared object file or if the routine name'
        PRINT, '      has a mixed-case name.'
        RETURN
ENDIF

; Get default directory
IF (N_ELEMENTS(dir) EQ 0) THEN BEGIN
        SPAWN, 'pwd', dir
        dir = dir[0] + '/'
ENDIF

IF (N_ELEMENTS(entry) EQ 0) THEN entry = shr_obj_name

; Place a '/' at end of directory name if missing
IF (STRMID(dir,STRLEN(dir)-1,1) NE '/') THEN dir = dir + '/'

linkimage,entry,dir+shr_obj_name+'.so',code,entry
        ; last parameter needed for mixed case routines

IF (not keyword_set(silent)) THEN $
        print,'Established linkimage for ', shr_obj_name

IF (entry NE shr_obj_name) THEN PRINT,'Entry Point: ', entry
```

```
return
end
```

Code Examples

Scalar Variables

We start with a procedure that takes a single floating point value, evaluates a simple quadratic equation, and returns this value in the original variable. It would be called from IDL as follows:

```
IDL> x = 2.0
IDL> quad1,x
IDL> print,x
      9.00000
```

```
#include <stdio.h>
#include <math.h>
#include "export.h"
        /* This header file contains all the IDL variable structures and

           functions necessary to write C module linkimages. */

void quad1(int argc, IDL_VPTR argv[])
        /* This C function is defined as "void" becaue it returns no
           value.  Rather the desired quantity is passed back through
           a command line parameter.  The "argc" and "argv" parameters
           are C constructs that access parameter values passed via
           the "command line". */

{

/* C variables */

float           x;      /* quadratic function argument */
float           y;      /* quadratic function value */


        x = argv[0]->value.f;
                /* Get the floating point value passed on the IDL
                   command line.  This statement could be replaced
```

```
            by: x = (*argv[0]).value.f which emphasizes that
            argv[0] is a pointer to the IDL variable structure
            (defined in Chap. 4). */

      y = 2*x*x - 3*x + 7;
            /* Evaluate the quadratic equation */

      argv[0]->value.f = y;
            /* Return this value to IDL through the first (and
               and only) parameter. */

return;
}
```

In this next example, we evaluate the same expression but return the value through a second IDL variable on the command line.

```
IDL> x = 2.0
IDL> quad2,x,y
IDL> print,x
     2.00000
IDL> print,y
     9.00000
```

```
#include <stdio.h>
#include <math.h>
#include "export.h"

void quad2(int argc, IDL_VPTR argv[])
{

/* C variables */

float          x;      /* quadratic function argument */
float          y;      /* quadratic function value */


      x = argv[0]->value.f;

      y = 2*x*x - 3*x + 7;


      argv[1]->type = IDL_TYP_FLOAT;
            /* Point to the field in the variable structure
```

```
                (of the second parameter "argv[1]") containing
                the data type and fill in with the single precision
                float value "IDL_TYP_FLOAT" defined in "export.h".
                Because this second parameter is on the command
                line, IDL has already allocated a variable
                structure for it. */

        argv[1]->value.f = y;
                /* Point to the field in the variable structure
                   containing the (float) value and load in "y" */


return;
}
```

Finally, we will show how to return the value through an IDL function.
This will show how scalar IDL variables are (dynamically) allocated.
The calling sequence is:

```
IDL> x = 2.0
IDL> y = quad3(x)
IDL> print,x
     2.00000
IDL> print,y
     9.00000
```

```
#include <stdio.h>
#include <math.h>
#include "export.h"


IDL_VPTR quad3(int argc, IDL_VPTR argv[])
                /* "quad3" is called as an IDL function and thus must be

                   declared as an IDL variable pointer because it
                   returns a value. */
{

/* C variables */

float           x;      /* quadratic function argument */
float           y;      /* quadratic function value */


/* IDL variables */
```

```
IDL_VARIABLE    *retdat;        /* "retdat" is declared as a pointer to
                                   an IDL variable allocated below. */

        x = argv[0]->value.f;

        y = 2*x*x - 3*x +7;

        retdat = (IDL_VARIABLE *) IDL_Gettmp();
                /* Call the IDL/C function "IDL_Gettmp" which allocates
                   memory for a scalar variable.  The pointer to this
                   new IDL variable is refered to by "retdat". */

        retdat->type = IDL_TYP_FLOAT;
                /* Point to the field in the IDL_VARIABLE structure
                   containing the data type and fill in with the
                   float value (IDL_TYP_FLOAT) defined in "export.h". */

        retdat->value.f = y;
                /* Point to the field in the VARIABLE structure
                   containing the (float) value and load in "y" */


return(retdat);         /* return the IDL_VARIABLE pointer "retdat" as
                           the value of the "quad3" function */
}
```

Array Variables

In the following example we show how to read in data from an IDL array, take its transpose, and then return this new array as (1) a procedure parameter and (2) a function value.

```
IDL> a = FINDGEN(2,3)
IDL> trans_pro,a,b
IDL> help,b
```

```
#include <stdio.h>
#include <math.h>
#include "export.h"

void trans_pro(int argc, IDL_VPTR argv[])
```

```
{

IDL_LONG        i, j, k;
IDL_LONG        inp_dim[2], out_dim[2];

/* IDL variables */

float           *input;         /* pointer to input array data field */
float           *outp;          /* pointer to output array data field */

long int        *dims;          /* pointer to array dimensions array*/
IDL_VARIABLE    *retdat;        /* pointer to output variable */


        input = (float *) argv[0]->value.arr->data;
                /* Get pointer to input array.  argv[0] points
                   to the "value.arr" field which in turn points

                   to the beginning of the input data.  The data

                   are declared as a character (byte) data type
                   in the IDL_VARIABLE structure definition.
Because
                   the data are to be interpreted as single
precision
                   floating point values, "input" must be "cast"

                   as a floating point pointer. */

        dims = (IDL_LONG *) argv[0]->value.arr->dim;
                /* Get input array dimensions.  "Dims" is
                   declared as a (long) pointer as thus points
                   to the first element of this vector array. */

        for (i=0; i<=1; ++i)
                inp_dim[i] = *dims++;
                /* Load input array dimensions.  Increment the
                   pointer after accessing each element in the
                   array. */

        for (i=0; i<=1; ++i)
                out_dim[i] = inp_dim[1-i];
                /* "Transpose" input array dimensions to give
                   output array dimensions. */

        outp = (float *)
                IDL_MakeTempArray((IDL_TYP_FLOAT,2,out_dim,
                IDL_BARR_INI_ZERO, &retdat);
```

```
                /* Call the IDL/C function "IDL_MakeTempArray"
to
                   allocate space for output array.  The first
                   argument gives the data type (single
                   precision float in this case), the second
                   gives the number of dimension of the array
                   (2), the third, the C array containing each
                   array dimension, the fourth gives the
                   initialization to be applied (zero all
                   elements), and the last gives the location
                   in memory of this IDL variable.  This
                   function returns the location in memory of
                   the first element of the actual data.  As
                   in the case above, it must be cast into the
                   proper type (float). */


        for (i=0; i<out_dim[1]; i++)
        {
                for (j=0; j<out_dim[0]; j++)
                {
                        k = i*out_dim[0] + j;
                        /* pointer offset of output array */

                        *(outp + k) = *(input + j*inp_dim[0] + i);
                        /* load (out)[i][j] with (in)[j][i] */
                }
        }


        IDL_VarCopy(retdat,argv[1]);
                /* Call to IDL/C function "IDL_VarCopy" to
transfer
                   "retdat" IDL_VARIABLE to second parameter
"argv[1]"
                    in command line.  Actually, only the pointer

                    to "retdat" is copied. */


return;
}
```

If written as an IDL function, the only changes would be to declare the function as a "IDL_VPTR" rather than as "void", eliminate the call to

"IDL_VarCopy", and replace the return statement with "return(retdat)".

```
IDL> a = FINDGEN(2,3)
IDL> b = trans_fun(a)
IDL> help,b
```

```
#include <stdio.h>
#include <math.h>
#include "export.h"

IDL_VPTR trans_fun(int argc, IDL_VPTR argv[])
{

IDL_LONG        i, j, k;
IDL_LONG        inp_dim[2], out_dim[2];

/* IDL variables */

float           *input;         /* pointer to input array data field */
float           *outp;          /* pointer to output array data field */

IDL_LONG        *dims;          /* pointer to array dimensions array*/
IDL_VARIABLE    *retdat;        /* pointer to output variable */


        input = (float *) argv[0]->value.arr->data;

        dims = (IDL_LONG *) argv[0]->value.arr->dim;

        for (i=0; i<=1; ++i)
                inp_dim[i] = *dims++;

        for (i=0; i<=1; ++i)
                out_dim[i] = inp_dim[1-i];

        outp = (float *)
                IDL_MakeTempArray(IDL_TYP_FLOAT,2,out_dim,
                IDL_BARR_INI_ZERO, &retdat);


        for (i=0; i<out_dim[1]; i++)
        {
                for (j=0; j<out_dim[0]; j++)
                {
                        k = i*out_dim[0] + j;
                        /* pointer offset of output array */
```

```
                *(outp + k) = *(input + j*inp_dim[0] + i);
                /* load (out)[i][j] with (in)[j][i] */
            }
        }


return(retdat);
}
```

The reader should note that the complete IDL variable is contained in "retdat". The pointer "outp", returned by the "IDL_MakeTempArray" function, refers to the data field within the IDL variable. Thus the "outp" variable is used when generating the transposed data array whereas "retdat" is used to return the complete IDL variable.

Strings

In the first example, we will access an IDL (scalar) string variable and print its length and content.

```
IDL> single_string, 'red'
```

```
#include <stdio.h>
#include "export.h"

void single_string(int argc, IDL_VPTR argv[])
{
UCHAR                *str;
IDL_STRING             string_structure;

string_structure = argv[0]->value.str;

printf("string length: %d\n", string_structure.slen);
printf("string: %s\n", string_structure.s);
```

```
return;
}
```

In the next example, we present a C function that will take an array of strings, convert them to upper case, and return them in a new string array. Because strings don't have a fixed length, the data field of an IDL string variable contains a string descriptor rather than the string data itself. This descriptor is a structure of length eight and is described in the External Development Guide. It contains the string length, the string type (null or allocated), and a pointer to the actual string data. IDL provides macros and function for accessing and storing strings which simplifies their handling. They are also described in the External Development Guide.

```
IDL> a = ['red','yellow','green']
IDL> print, upper(a)
```

```
#include <stdio.h>
#include <string.h>     /* string function header file */
#include <math.h>
#include "export.h"

IDL_VPTR upper(int argc, IDL_VPTR argv[])
{

short               i,j,n_str;
UCHAR                   str_vec[32];
UCHAR                   *str;


/* IDL pointer variables */

IDL_STRING              *lc_ptr;        /* pointer to lower case input
                                data descriptor */

IDL_STRING              *uc_ptr;        /* pointer to upper case output
                                data descriptor */

unsigned short          str_len;        /* string length variable */
```

```
IDL_VARIABLE          *retdat;       /* pointer to IDL output
                           variable */


      lc_ptr = (IDL_STRING *) argv[0]->value.arr->data;
                  /* Cast pointer to input string data
                     descriptor as IDL_STRING data type.  Remember

                     that the data field of an IDL array is
                     declared as a (generic) character pointer.
                     The cast operator thus associates the string
                     descriptor structure with the pointer. */


      n_str = argv[0]->value.arr->n_elts;
                  /* Get number of elements in string array */

      uc_ptr = (IDL_STRING *) IDL_MakeTempArray(IDL_TYP_STRING,
                        argv[0]->value.arr->n_dim,
                        argv[0]->value.arr->dim,
                        BASICARR_INI_NOP, &retdat);
                  /* Allocate output string array variable using
                     parameters of input variable and cast the
                     returned data pointer as a IDL_STRING
variable. */


      for (j=0; j<n_str; ++j)
      {

            str_len = lc_ptr->slen;
                  /* Get string length from current element of
                     string descriptor array. */

            str = IDL_STRING_STR(lc_ptr);
                  /* Get pointer to actual string.  Use of
STRING_STR
                     avoids problems that can occur when accessing

                     zero length strings.  Note: "lc_ptr" points
                     to an element in the string descriptor; "str"

                     points to the string itself. */

            for (i=0; i<=str_len; ++i) str_vec[i] = toupper(*str++);

                  /* For each character in the string, change to
```

```
                upper case and store in "str_vec" character
                array. */

        IDL_StrStore(uc_ptr,str_vec);
                /* This IDL/C macro "stores" the "str_vec" array

                into the IDL output string array.  "Storing"
                consists of filling in the output STRING
                structure for each element of the output
                string array. */

        lc_ptr++;
        uc_ptr++;
                /* Increment the pointers to both the input and
                output string arrays. */


    }


return(retdat);         /* return IDL output variable */
}
```