A Dynamically Linkable Graphics Library

Diomidis D. Spinellis
Department of Computing, Imperial College of Science and Technology, 180 Queens Gate, London SW7 2BZ, U.K.

March 1988

DRAFT

A Dynamically Linkable Graphics Library

Diomidis D. Spinellis
Department of Computing, Imperial College of Science and Technology, 180 Queens Gate, London SW7 2BZ, U.K.

SUMMARY

The design issues behind the implementation of an efficient and portable graphics library are discussed. A description of its components is given and the constraints leading to dynamic linking are presented. Techniques allowing the transparent dynamic linking of library elements are analysed and two implementations of a system that automatically creates dynamically linkable code are presented. The one implementation is based on traditional UNIX tools and the other on the perl programming language. The two implementations are compared.

KEY WORDS : Dynamic linking Graphics libraries Perl

INTRODUCTION

During the design of an interactive graphics pre- and post-processor for a finite element analysis system, the problem of portably displaying the output on a wide variety of graphics output devices was encountered. The program, initially, had to run on IBM-PC class machines running the MS-DOS operating system. In a latter stage it was ported to run under the UNIX operating system on Sun and microVAX workstations.
The program is used to inspect structures represented by wire frames containing hundreds of elements in two distinct phases. First, before input to the finite element analysis program, the wire frame is examined in order to visually verify its form. After the analysis the program is used to inspect the distortions suffered under specific loads. The user may rotate the structure in three dimensions, view specific parts of it, label its joints and members and perform various other operations on it. The interactive nature of the program and the range of machines it was designed to operate on, made its design focus on a fast implementation. The main program consists of about 7000 lines of code written in the C[1] programming language.
MS-DOS does not provide an application graphics interface and the ROM *Basic Input Output System* (BIOS)[2] that is available on these machines does not support devices other than those manufactured by the machine vendor. In addition the functions it provides are minimal. Typical functions could display a character, set a point to a specified colour and set up the

screen for a particular graphics mode. The implementation of more advanced commands using the BIOS would suffer from the overhead of calling it for every point drawn. Some experimenting using its functions demonstrated that the speed of a program using it, was clearly unacceptable for interactive use.
The use of a graphics platform like GKS[3] or Microsoft Windows[4] was considered. GKS offers device independent graphics functions and would offer the portability desired. Microsoft Windows is a window oriented user interface. Applications written using the functions it provides are portable among a range of hardware as it is responsible for implementing the input/output functions for a particular device. The use of those programs was not opted for, because they were not widely available among the user base. Particularly the use of a Microsoft Windows interface would burden the application with one more level of non-portability to machines not running MS-DOS.

THE LIBRARY APPROACH

The initial approach to the problem was the isolation of the graphics functions from the main program in order to achieve portability. In addition to that some useful, but not device specific graphics functions were identified. Thus two function libraries were created. The one consists of all device specific functions and the other of device independent ones.
The criterion used to isolate the functions in those two groups was mainly the efficiency with which a particular function could be implemented using other device specific functions. Furthermore device specific functions should only do the absolute minimum work that was device specific. Non device specific functions would provide better interfaces. This was decided in order to simplify the implementation of the graphics library on a wide variety of graphics devices.
Using this criterion functions like draw-a-line and display-a-character were immediately put in the device specific library. An example of a function that was put in the device dependent library only because it could be implemented very efficiently in a device specific way was the crosshair function. This function allows the identification of a screen point by the intersection of two lines at right angles, covering the whole screen. The implementation should be very efficient as this function is usually driven by a mouse forming a part of the user interface. Efficient implementations for this function are device specific as they can range from manipulating whole words, for the horizontal line in bit mapped displays, to invoking device specific functions, as for the Tektronix[5] type displays.
Apart from the functions provided a number of global variables was also included in the device specific part of the library. These specified various display characteristics such as the screen size and aspect.
All the portable library is written in C and its size is about 1500 lines. Some of the device specific libraries are written in C, but others have parts written in assembly language. Their size

varies from 140 lines which is the library interfacing to the graphics library provided by the compiler vendor to 1300 lines which is the size of the EGA[6] interface library. Up to now 12 different libraries have been developed.

Functions provided

The device specific part

The functions that the device dependent library provides are listed in table 1. Table 2 outlines the global variables that are made available.

Name	Function
cls	Clear the screen.
disp	Display a given graphics page.
gmode	Set graphics mode in effect.
gpage	Set graphics page to be used for output.
grpoint	Graphics cross hair pointer.
horxor	Invert a character row.
line	Draw a line.
locate	Change the cursor position.
plot	Plot a point on the screen
point	Return the colour of a point on the screen.
scr_get	Save the screen buffer to memory.
scr_put	Restore the screen buffer from memory.
tmode	Set text mode in effect.
win_get	Save an area of the screen to memory.
win_put	Restore an area of the screen from memory.
win_size	Return memory required for a screen area.
wrt	Write a character attribute on the screen.
wrtrep	Write a character a number of times.

Table 1

Name	Function
aspect	The screen aspect ratio.
cols	The screen size in columns.
rows	The screen size in rows.
scr_can_do	The availability of the scr_ functions.
scr_planes	The number of screen planes.
scr_size	The size of one screen plane in bytes.
xpels	The number of pixels in the screen x coordinate.
xsize	The width in pixels of one character.
ypels	The number of pixels in the screen y coordinate.
ysize	The height in pixels of one character.
win_can_do	The availability of the win_ functions.

Table 2

Not all functions are supported by all hardware specific libraries. Some may be just stubs in a particular implementation. Where this can affect the program function, variables specify whether a function is available for a specific configuration. The functions were initially designed as part of the three dimensional view program and based on the functions provided by the IBM-PC BIOS. The interface design strongly reflects this fact. The function interface is efficient as used in that program and relatively easy to implement in machines with PC BIOS support. For efficiency reasons parameters that tend to stay the same between a number of calls to functions (such as the output screen page) are specified by a different function in order to avoid the argument passing overhead. On the other hand parameters that would change from one function call to another are arguments of one function in order to minimise the function calling overhead. Thus the sequence of calls needed for drawing a line in a specific colour are not :

```
SetDrawColour( colour ) ;
SetDrawingFunction( copy ) ;
MoveTo(x1, y1) ;
DrawTo(x2, y2) ;
```

A single function provides this interface :

```
line(x1, y1, x2, y2, colour ) ;
```

The interface is thus less general and focused towards a specific application. However the library has been successfully used in the design of other packages without any serious problems.
A more serious design error is the name specification of the routines which results in considerable name space pollution in flat naming programming systems such as the C programming language. The choice of some function names can also be described as unfortunate (this applies to the device independent library as well).

The portable part

The functions that are currently provided by the device independent library are listed in table 3.

Name	Function
accept	Get user input at a specific screen position.
circle	Draw a circle.
displayf	Display formatted data at a specific screen position.
horchar	Repeat a character horizontally on the screen.
hpick	Pick an item from a horizontal menu, providing help.

lineclip	Draw a line, clipping parts outside screen limits.
mpick	Pick an items from a table, providing help.
text	Draw text using vector fonts.
userin	Get input form the user allowing line editing.
verchar	Repeat a character vertically on the screen.
vpick	Pick an item from a vertical menu.
wclose	Close a window.
wdel	Delete a window from the screen.
wdrag	Change the position of a window.
wleave	Leave a window.
wopen	Open a new window.
wscroll	Scroll the contents of a window.
wsize	Change the window size.
wuse	Use a window.

Table 3

No global variables are defined by the library. The functions provided rely on functions from the device specific library. The dichotomy of the two libraries was established gradually and in the early phases of the development functions tended to migrate from one to the other, as it was tried to create a balance between efficiency and ease of implementation.
The windowing functions provided, form a rather crude windowing system and have not been extensively used. With the constantly increasing use of many different windowing environments a portable windowing library is under consideration.
Two families of functions provide character output. Raster fonts are used for speed and vector fonts for efficiency. Naturally all user interface functions such as menus and input procedures use the raster font.

Initial implementation

The initial implementation of the graphics library was in the form of two MS-DOS object libraries. The program was linked with the device independent library and one of the device dependant ones. No recompilation was needed for different devices. This method had the drawback of producing a different executable module for every device.

DYNAMIC LINKING

Design constraints

As the number of devices for which libraries were created increased the drawbacks of the separate linking became apparent. More and more different executable programs had to be stored, distributed and maintained. The testing of those programs became

difficult as a different linking session was needed for every executable module. At that point the quest for alternative solutions began.
For producing an alternative method the following constraints were set :

a) One executable program should be able to run on any graphics hardware configuration.
b) The original, library based, interface should still be a valid linking option.
c) No changes to the program should be required.
d) The solution should draw upon the resources of the *compiled* device specific libraries.
e) No performance penalties should be paid.
f) The solution should be part of the application program and not part of the operating system or environment.

In addition to the constraints noted above it should be noted that the operating system does not offer any support for shared or dynamically linked libraries (it is a feature of OS/2 though), no memory protection is available and code is stored in the same type of memory as data.

Module decomposition

Constraint a) hinted the solution of a device specific driver that would be loaded by the application program at run time. Due to constraints f) and e) a special output format that would be interpreted by a filter or a resident device driver was not opted for.
The constraints b), c) and d) set the framework for the following solution : The library code was divided into two parts. A stub part would resolve all references during the linking phase by providing dummy procedures and global variables. These procedures would, when called, initialize the driver by loading device specific code, replace the reference to themselves with a reference to the real routine and repeat the original call.
The format of the device specific code is ordinary relocatable executable code in the same format as that used by executable programs. The operating system provides a function to load such a file into memory performing relocation of entries. The first items in the file are the locations of all functions and global variables. That file could be generated just by linking one of the device specific libraries with a small assembly program containing the initial table and references to all the functions. Thus no specific linking tools needed to be implemented.

Call implementation

The re-execution of a function call after patching the original code with a new address was tricky and the end code is not something one should be proud of. The code is highly compiler specific, but it survived three consecutive releases of the compiler. The example that follows is the stub function for *cls*.

```
void cls(ab)
     int ab ;
{
     char ** caller = (char **)((char *)&ab - sizeof( char * )) ;
     if( ! init_drv )_init() ;
     *(char **)(*caller - sizeof(char *)) = addr[11] ;
     *caller -= sizeof(char *) + 1 ;
}
```

The argument *ab* is not an argument that is passed to *cls*. It is used to find the location of the return address in memory. As *ab* is the last argument pushed onto the stack before the function call (or at least that's what the compiler thinks) its address is one pointer location after the return address of the function. Having the return address into the variable *caller*, the need for driver initialisation is checked. If initialisation is needed it is performed. After initialisation the addresses of all external references are placed in the array *addr*. Here *addr[11]* is the address of the *cls* routine in the code that was loaded. The next line sets the address used by the call instruction to the address of the real *cls* routine and after that the return address is adjusted so that the call will be repeated. From then on every time that instruction is encountered the real *cls* function will be called instead of the stub one.

Static data implementation

Having solved the problem of integrating the graphics functions into the main instruction stream one more problem needed solving. That was the provision of data space for the graphics functions to use. As the graphics functions were separately linked from the main program no information was available on the memory size required by them and thus the memory references to the main programs static variables and those of the graphics functions overlapped.
The way this problem was solved was by noticing that because the graphics functions were not linked with any other code the only static data they required was allocated at the start of the *DATA* segment. Furthermore by explicitly initialising all variables to a value the additional complication of catering for data in the *BSS* segment was eliminated. Thus after linking all different device dependent drivers the linker output maps were examined, the largest amount of static data required was noted and another object module to be linked to the main program was created. That module contained a new *NULL* segment (which is the segment

immediately before the *DATA* segment) whose contents were the ones supplied by the compiler (library copyright message and a checksum) plus empty space for the graphics functions static data.
As the maximum amount of static data required by a function was about 30 bytes this was no an inefficient implementation. It is not very clean however, as it includes some code from the compiler runtime library.

Prototype solution

With the way to implement the dynamically linkable device drivers designed and tested, a way was needed to automatically create all the modules required for such a library. Specifically such a system should take a library description and a set of device dependent libraries as input and produce code and data object file stubs and dynamically linkable device driver files as output as seen in figure 1.

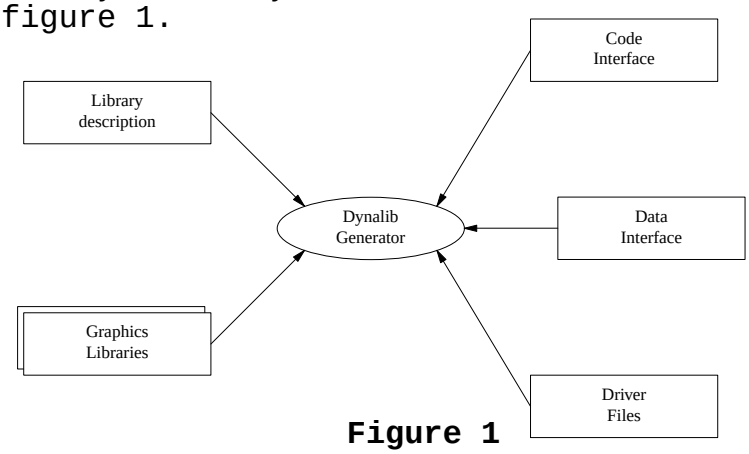

Figure 1

It was decided to use standard UNIX tools to create such a system. The tools used were awk[7] and sed[8]. The library description file format was defined to be composed of two parts. In the first part the various libraries are described. For every device specific library, its name and the location of a file that can be used to resolve are references are given. The name, is used for naming the driver file. Following that list comes the word ENDLIB and then a list of all symbols that compose the library. After each symbol comes its type which can be one of ‘*word dword qword far*’. These indicate the size of the symbol if it is a variable, or that it is a function. Currently pointers are not supported. The following is a fragment from the original specification file :

```
.............
cga lcga.lib
mcga lmcga.lib
ega lega.lib
herc lherc.lib
vga lvga.lib
wy700 lwy700.lib
```

```
ENDLIB
aspect qword
cols word
scr_planes word
cls far
disp far
.........
```

This file is input to an awk script. The awk script generates an assembly file, which when linked with a particular graphics library will create a driver file, the stub code file to be linked with the main program in C and a command file to be executed in order to do the actual linkings required. The command file also contains instructions that will pipe the output from linking the particular driver file through a sed script in order to generate the data binding.
Three awk arrays are used to store the names and types of all variables referenced in the library description file. For every function found in the file a stub function is output in the C file. If a variable reference is encountered, a global variable definition for the particular size is emitted and the name of the variable is stored in an array used for the particular type. This is a code fragment that generates references to *double* variables :

```
start == 1 && /[  ]+qword[  ]*$/ {
     initqword[ addr++ ] = $1 ;
     printf "double %s ; ", $1 >>cfile ;
     initqwords++ ;
}
```

The variable *start* holds state information needed for awk to know which part of the library description file it is processing. When the end of file is reached the code for the initialising function is generated.
The initialising function finds the size of the driver and allocates memory to store it using malloc. It then loads the driver code into memory, initialises the pointer *addr* which contains the addresses of all elements to point to the correct part of the driver and then outputs code to initialise the global variables in the stub module to the values found in the real module. Again an example for the *double* variables follows :

```
if( initqwords )
     for( i in initqword )
          printf " %s = *(double *)addr[%d] ; ",
                 initqword[i], i >>cfile ;
```

The assembly file that is linked to a library in order to produce the driver file is generated on the fly. Every line that specifies an entry is emitted with ‘*extern* _’ perpended to it and followed by a line ‘*dd _name*’ where *name* is the name of the function or variable.

The processing of the initial driver names and library descriptions generates the command file that takes control after this point is reached. That command file links the assembly file generated with each library and renames the result to the driver name specified. The linker map output is passed through sed in order to create an assembly file that will create a *NULL* segment with extra length equal to the maximum data segment length encountered. The sed command that does this is :

```
/^ [0-9A-F]*H [0-9A-F]*H [0-9A-F]*H _DATA/s/^ [^H]*H [^H]*H
\([^H]*H\).*$/IF \1 GT DATALEN
DATALEN = \1
ENDIF
/p
```

For every data segment length is encounters it generates a sequence of commands of the type :

```
IF   length GT DATALEN
DATALEN = length
ENDIF
```

where *length* is the length of the particular data segment. This sequence is recognised by the macro assembler and as a result at the end of the final sequence DATALEN is set to he maximum length required.

Unified solution

From the above description it is evident that the whole system used to generate the drivers although working is not a good example of nice code. Its interfaces are unclear and much of the work is done in an non-obvious and highly involved way. Although the system was used for a period of six months in order to develop more screen drivers the author always felt that a better implementation was needed.
During February 1988 Larry Wall released on the USENET the source for a programming language called perl (Practical Extraction and Report Language)[9]. According to the manual page perl was supposed to combine the best features of C, sed, awk and sh. It was felt that perl was an ideal language to implement this system. The rewrite process was easy and at the end the three (plus one intermediate) file original system was reduced to one perl script. The size of the system was reduced by 25% and the execution speed increased by 31%.
The more important results were gains in the overall quality of the system. By putting the whole system into one file its interfaces became clear and the way it functioned obvious. This may seem as a paradox, taking into account the rationale behind modular implementations. The reason why the system broken into parts was worse than one program, is that the breaking into parts was not directed by a functional decomposition, but by deciding

what parts of the system could be managed by a particular tool. The maximalist design philosophy behind perl clearly helped to improve the quality of the finished design.

CONCLUSIONS

Although the initial design of the library was designed to be used on a specific hardware device and by one program over the last two years this has changed. Currently 12 different output devices or protocols are supported ranging from the PS/2 VGA adapter to an X Windowing System[10] interface. The programs that have used it are a three dimensional perspective view system, a star chart display program and a specialised CAD program.
The library worked well for the scope it was designed for. It provided a portable interface across many hardware configurations. The dynamic linking capability was found to be easy and efficient to use and resulted in better and more portable programs. With the ever increasing use of windowing systems the library design has started to affect the usability of the programs that depend on it. User friendliness and conformance with a particular windowing, presentation and user interface standard are compromised in order to increase portability. This is an issue that has to be addressed in the future and there are already steps towards a solution.
Dynamic linking is an interesting possibility in application areas with diverse equipment and/or user requirements. The technique described is almost totally transparent to the application programmer. It can be extended to allow users or OEMs to create their own drivers. Currently experimentation is being made with input libraries and language specific drivers.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, ‘The C Programming Language’, Prentice-Hall, 1978.
2. ‘IBM Personal Computer, Technical Reference, First Edition’ (Revised May 1983).
3. F. R. A. Hopgood and D.A. Duce, ‘Graphics Standards - The Current State’, August 1986.
4. Microsoft Corporation, ‘Microsoft Windows, Operating Environment, Users Guide’, 1985.
5. ‘Tektronix, 4014 and 4014-1 Computer Display Terminal’, 1974
6. ‘IBM Enhanced Graphics Adapter, Technical Reference Manual’, August 2, 1984.
7. A. V. Aho, B. W. Kernighan and P. J. Weinberger, ‘Awk - A Pattern Scanning and Processing Language,(Second Edition)’, September 1, 1978.
8. L. E. McMahon,‘SED - A Non-interactive Text Editor’, August 15, 1978.
9. L. Wall, ‘Perl - Practical Extraction and Report Language’, March 15 1988.

10. J. Gettys, R. Newman and T. D. Fera, ‘Xlib - C Language X Interface’, November 16, 1986.

APPENDIX

The dynamically linked library generator in perl

```
@REM=("
@perl %0.bat %1 %2 %3 %4 %5 %6 %7 %8 %9
@end ") if 0 ;

# Create a driver interface
# (C) Copyright 1987,88 Diomidis Spinellis. All rights reserved.
# See driver.doc file for info.
# The fast option produces only the .drv files and skips the compiling
# process. It can be used when only the code of a driver has changed.

# Generate assembly batch and C file
# Pass 1 of mkdrv (create a driver)

$#ARGV == 0 || die "$0 : Usage $0 file" ;

open(infile , '<'.($drivername = $ARGV[0]) ) || die "Unable to open $drive
open(asmfile , '>'.($asmfilename = ( $ARGV[0].'.asm') ) ) ;
open(cfile , '>'.( $cfilename = ('c'.$ARGV[0].'.c') ) ) ;
print asmfile "
;Automaticaly generated assembly file
; Prohibit null pointer assignments by leaving space for NULL sgement
NULL SEGMENT   PARA PUBLIC 'BEGDATA'
     db   37h dup(?)
" ;
print cfile "
/*Automaticaly generated C file */

#include <stdio.h>
#include <dos.h>
#include <malloc.h>
#include <errno.h>

#define NULLLEN 0x37


static int init_drv = 0 ;
static char **addr ;
static void drv_init() ;
extern int errno ;

" ;
# Process all the library info
while( ($_ = <infile>) && ($_ ne "ENDLIB " ) ){
     ($name, $lib) = split ;
     push( names, $name ) ;
     push( libraries, $lib ) ;
```

```
}

# Check for EOF
if( $_ ne "ENDLIB " ){
     unlink $asmfilename, $cfilename ;
     die 'EOF reached before ENDLIB' ;
}

# Process external symbols
while( <infile> ){
     ($name, $distance) = split ;
     push( externs, sprintf(" extrn _%s : %s ", $name, $distance ) ) ;
     printf asmfile " dd _%s ", $name ;
     # Have the caller call the new function and redo the call
     if( $distance eq 'far' ){
          printf cfile "
void %s(ab)
     int ab ;
{
     char ** caller = (char **)((char *)&ab - sizeof( char * )) ;
     if( ! init_drv ) drv_init() ;
     *(char **)(*caller - 4) = addr[%d] ;
     *caller -= 5 ;
}

", $name, $addr++ ;
     } elsif( $distance eq 'word' ){
          # Store variable occurances in an array to create init()
          $initword{ $addr++ } = $name ;
          printf cfile "int %s ; ", $name ;
          $initwords = 1 ;
     } elsif( $distance eq 'dword' ){
          $initdword{ $addr++ } = $name ;
          printf cfile "long %s ; ", $name ;
          $initdwords = 1 ;
     } elsif( $distance eq 'qword' ){
          $initqword{ $addr++ } = $name ;
          printf cfile "double %s ; ", $name ;
          $initqwords = 1 ;
     }
}

# The end of the cfile (Driver initialisation part)
printf cfile '
static void drv_init()
{
     FILE *f ;
     int codelen, flen ;
     int headerlen ;
     char *codep ;
     union REGS srv ;
     struct SREGS segs ;
     static char *name = "%s.drv" ;
     struct {
```

```
          int segmem ;
          int reloc ;
     } pblock , *pblockp = &pblock;


     init_drv++ ;
     /* Calculate length of code */
     if( ( f = fopen(name,"rb") ) == NULL ){
          perror("Erron in opening driver file %s.drv") ;
          exit( 2 ) ;
     }
     fseek(f, 8l, 0) ;
     fread(&headerlen, sizeof(int), 1, f ) ;
     fseek(f, 0l , 2 ) ;
     flen = (int)ftell( f ) ;
     fclose( f ) ;
     codelen = flen - headerlen*16 ;
     if( ( codep = malloc( codelen + 16 ) ) == NULL ){
          perror("Out of memory for driver storage") ;
          exit( 2 ) ;
     }
     /*Allign codep on a paragraph boundary and zero offset*/
     codep = (char *)( (long)(FP_SEG(codep)+(FP_OFF(codep)>>4)+1)<<16 ) ;
     /*Load overlay*/
     srv.h.ah = 0x4b ;
     srv.h.al = 0x03 ;
     srv.x.dx = FP_OFF( name ) ;
     segs.ds = FP_SEG( name ) ;
     pblock.segmem = pblock.reloc = FP_SEG( codep ) ;
     segs.es = FP_SEG(pblockp) ;
     srv.x.bx = FP_OFF(pblockp) ;
     intdosx(&srv,&srv,&segs) ;
     if( srv.x.cflag ){
          switch(srv.x.ax ){
          case 1 :
               errno = EINVAL ;
               break ;
          case 2 :
               errno = ENOENT ;
               break ;
          case 8 :
               errno = ENOMEM ;
               break ;
          default :
               errno = EZERO ;
               break ;
          }
          perror("Error in loading driver") ;
          exit( 2 ) ;
     }

     addr = (char **)(codep + NULLLEN ) ;
', $drivername, $drivername ;
```

```
# Initialize variables (if neeeded)
if( $initwords ){
     while( ($address, $name) = each( initword ) ){
          printf cfile " %s = *(int *)addr[%d] ; ", $name, $address ;
     }
}
if( $initdwords ){
     while( ($address, $name) = each( initdword ) ){
          printf cfile " %s = *(long *)addr[%d] ; ", $name, $address ;
     }
}
if( $initqwords ){
     while( ($address, $name) = each( initqword ) ){
          printf cfile " %s = *(double *)addr[%d] ; ", $name, $address ;
     }
}

print cfile " } " ;

print asmfile "
NULL ENDS

_DATA     SEGMENT   WORD PUBLIC 'DATA'
_DATA     ENDS

_BSS SEGMENT   WORD PUBLIC 'BSS'
_BSS ENDS

CONST     SEGMENT   WORD PUBLIC 'CONST'
CONST     ENDS

DGROUP    GROUP     NULL,_DATA,_BSS,CONST
" ;
for( $i = 0 ; $i <= $#externs ; $i++ ){
     print asmfile $externs[$i] ;
}
print asmfile "
public    __acrtused          ; To resolve external refs
__acrtused = 9876h       ; Funny value not matched by CV
     END

" ;

close cfile ;
close asmfile ;

system ('masm', '/Mx', $asmfilename, ';') ;

$datalen = 0 ;
for( $i = 0 ; $i < $#names ; $i++ ){
     if( $libraries[$i] =~ /.[lL][iI][Bb]/ ){
          $command = sprintf ('link /MAP /NOI %s,%s,con,%s|', $drivername,
     } else {
```

```
          $command = sprintf ('link /MAP /NOI %s+%s,%s,con;|', $drivername
     }
     open( linkout, $command ) ;
     while( <linkout> ){
          if( /_DATA/ ){
               @dataline = split( /[   H]+/ ) ;
               if( hex( $dataline[3] ) > $datalen ){
                    $datalen = hex( $dataline[3] ) ;
               }
          }
     }
     close( linkout ) ;
     unlink( '/lib/'.$names[$i].'.drv' ) ;
     rename( $names[$i].'.exe', '/lib/'.$names[$i].'.drv' ) ;
}
open( dasmfile, '>d'.$drivername.'.asm' ) ;
print dasmfile "
NULL SEGMENT   PARA PUBLIC 'BEGDATA'
     db   8 dup(0)
     db   'C Library - (C)Copyright Microsoft Corp 1986'
     db   01fh,0,0,0
     db   $datalen dup(?)
NULL ENDS
DGROUP    GROUP     NULL
     END
" ;
close( dasmfile ) ;
system ('masm', '/Mx', 'd'.$drivername, ';' ) ;
unlink ('/lib/d'.$drivername.'.obj' ) ;
rename ('d'.$drivername.'.obj', '/lib/d'.$drivername.'.obj' ) ;
unlink $asmfilename, 'd'.$asmfilename, $drivername.'.obj' ;
exec ('cc', '-Zi', '-c', '-Fo/lib/c'.$drivername, '-AL', $cfilename ) ;
#This will never happen. I do an exec because perl and c2 can't coexist
unlink $cfilename ;
```