The Link
Between C and Assembly

David Godshall
Elkhart, Indiana

...How would you like to be able to access a machine language routine the same way you would access a C function?

If you own "Power C" from Spinnaker, you own an excellent and powerful implementation of the C language. If you don't, but have been thinking about buying a copy, the possibilities opened by this article may be enough to push you over the edge. Even if you don't own and don't plan to buy Power C, the techniques mentioned in this article may apply to other compilers that compile in two separate stages (from source into object, then from object into executable) or even for other types of programs.

The Problem...

I was writing some graphics routines to be used from C programs. I decided that, while Power C is probably the fastest compiler for the C-64 ("A Comparison of Language Speeds", Volume 7, Issue 5), nothing can beat hand-coding a piece of code for speed. I promptly set about coding assembly language routines to clear the bitmap, plot points, and all the other nice things you like to do to graphics screens quickly. In looking for a good way to use these routines from my C programs, I came across the SYS function. I tried it. I gave up on it. The main trouble with the SYS function is that it is little better than the BASIC SYS command. It assumes the code is already in memory (if it isn't, a special disk access is needed to put it there). It does not provide very descriptive access to the routines. The one thing it has over BASIC's SYS command is that it can handle parameters - three bytes' worth. Some of the functions I wanted to access, however, required more than 24 bits' worth of information.

So, giving up on the SYS function, I looked around and about and inside out for a better way. And I found it! My search led me to the internals of the object files - those mysterious files that are the limbo between pure source code and pure executable machine language. I didn't have a lot of hope. I've looked in real spaghetti files before and was afraid the object files would turn out to be too complicated for me to figure out without the aid of extensive documentation and weeks of personal interviews with Power C author Brian Hilchie. My fears were groundless.

Starting My Quest

To begin with, I already sort of knew what the linker does. It takes an object file, moves it to a specific address in memory, and then checks to see what the file needs to be linked with. It then gets those files, puts them at unique addresses, and checks what files they need. Finally, when all the files are linked in with everything they need and the linker can't find anything else to add, it writes the executable file containing all the object files all linked up nicely. But what is in an object file? How does it let the linker know what it wants and what it has to offer?

My first clue to the general format of the object files came from the mysterious and totally undocumented (in my manual, at least) LIB.C file. This is a nice utility that puts a list of all the global identifiers from multiple object files into one file so that the linker can find their functions and variables very easily. The first four files on the library side of the disk (STDLIB.L, STDLIB2.L, SYSLIB.L and SYSLIB2.L) can be examined and modified with this utility.

By looking at how LIB.C scans object files, I was able to determine that object files are divided into four sections. I call them the Code section, the Relocate section, the Global section, and the External section. Each section begins with a word (two bytes in low/high order) indicating how long the section is in bytes (for the Code section) or entries (for the other sections). The general format is summarized in Table 1.

Section 1: The Code Section

The first section looks familiar when viewed through a machine language monitor. It is straight machine language. Well, almost straight. There are a few differences that will be straightened out by the linker.

To begin with, there is some information that isn't known at the time the code is created. Any instruction referencing external functions or variables is going to have to have its operand filled in by the linker, so it doesn't matter what value is in the operand.

Secondly, one of the jobs of the linker is to decide where in memory to place the code. In order to enable the linker to do that easily, the code is generated by the compiler as if it were assembled to location $0000. In other words, if somewhere in the code you had a JMP instruction to transfer execution to the first instruction in the code section, it would be a JMP $0000 instruction. The linker can then relocate the code by calculating a new base address and adding it to the offsets contained in the operands of any instructions referencing a part of the code. But how does the linker know which instructions need to be adjusted for the new address and which are already pointing at the correct location (i.e. a ROM routine, a zero page location, etc.)? This is where the second section comes in:

Section 2: The Relocate Section

This section points out to the linker which instructions need to be adjusted during the address relocation process. If, for example, the instruction JSR $0073 is flagged by this section and the linker decided to relocate the code to base address $1153, then the instruction will be changed to JSR $11e6. If the JSR were not flagged by this section, if would remain JSR $0073.
This section consists of a list of addresses (as offsets from $0000) of instructions that need their operands relocated. The length word indicates how many addresses (of two bytes each) are in this section.

Section 3: The Global Section

This section tells the linker what the module has to offer to other modules. Any functions or variables that may be used by external routines are flagged in this section.

The length word indicates how many entries are in this section. Each entry will contain a name or identifier, a byte flag, and an address.

The name will be the one or more characters by which this variable or routine can be accessed. Remember that C is case sensitive and that identifiers coming out of the C compiler will be truncated to 8 characters. Terminate the name with a NULL (chr$0(0)).

The byte flag tells the linker whether the entry is referring to a location in the code section or an absolute location. If the byte is a one, then the linker will know that it is referencing the code section and will adjust the address when the code section is relocated. A zero tells the linker that the address does not need to be relocated (it may be pointing to a ROM routine or some other stable location).

Section 4: The External Section

This section is sort of the opposite of the global section. It tells the linker what external routines and variables are needed by this module. It contains entries similar to those in the global section. Each entry consists of the name of the routine or variable to link in, a word specifying how to link it in (offset), and the address of the instruction accessing the external entity.

The word following the name allows several possibilities for linking in the address of the external entity. First of all, it allows you to link in the address of the entity, the address plus one, the address plus two, etc. You can add up to 8191 bytes to the address. Secondly, you can decide to either link in the whole address (for absolute instructions such as LDA xxxx) or just the high or low byte of the address (for immediate instructions such as LDA *<xxxx or LDA *><xxx).

The way to specify these options is to take the number of bytes you want to add to the base address and multiply by four (to shift it into the upper 14 bits). Then you add 0, 1, or 2 depending on whether you want the whole address, the high byte, or the low byte respectively. The resulting value would go in this offset word.

Finishing up

To finish up the object file, just terminate it with two NULLs. Now you can give it the linker test! Beware, because the linker was created to link together modules created by the C compiler. Since the linker knows what type of object file the compiler is capable of creating, it isn’t very error tolerant and will lock up just about any irregularity. If you say there are five global entries, make sure you include exactly five. Make sure you terminate all identifiers with NULLs and the file with two NULLs. Et cetera.

Special Routines

There are several special external routines you may need to use when writing code to be linked in to work under the C environment. First is the CSTART routine. This routine is included in every C program and is responsible for setting up the C environment. It does some setup work, calls the main() function, then does some cleanup work before returning control to the shell or BASIC. Thus, CSTART must be the first thing to be called. The first instruction of the first file to be linked in must call this routine. But how do you know which of several files will be linked in first? To solve this, the C compiler puts a JMP CSTART instruction as the first instruction in every module it generates. If there is a chance that your module might be linked in first, you would also want to put in a jump to the CSTART external routine as the first instruction in your module.

Another important routine you will want to use is the CFUNCT_INIT routine. This is a routine that would be called first thing in any function you create. Normally, C functions call the routines CS105 on entry and CS106 just before returning instead of CFUNCT_INIT. CS105 copies the local variables (locations $2b-$4a) and parameters (cassette buffer $033c-$03fb) out of the way so the space can be used for new variables and parameters; CS106 copies them in again upon completion of the function. These require a lot of overhead, so the CFUNCT_INIT routine comes in handy for small routines that will not need to use the local variable area (they can use the temporary locations $22-$2a and $4b-$60) and that will not call other routines that will use the variable area or the parameter area.

Unfortunately, to explain CS105 and CS106 in more detail would take us out of the scope of this article and into memory management.

Parameter Passing

One of the advantages of linking machine language routines through the object file as opposed to the SYS function is the ability to pass dozens of parameters. On the originating end, the values of the variables you are passing (or their addresses, if you are passing pointers) are stored in memory starting at $033c and in the same order as they were declared in the function descriptor. The accumulator is then set to reflect the number of bytes used up by the parameters, and the new function is called; it can then access the parameters directly from this memory. As an example, the function

FRED (Age, Name, Weight, Height); where Age is a character type, Name is a pointer to an array of characters, Weight is a floating point number, and Height is an integer; would store:

| $033c - Age (one byte) |
| $033d - Name (low byte of pointer) |
| $033e - Name (high byte of pointer) |
| $033f - Weight |
| $0340 - Weight |
| $0341 - Weight (FP representation) |
| $0342 - Weight |
| $0343 - Weight |
| $0344 - Height (low byte) |
| $0345 - Height (high byte) |

The accumulator would be set to 10. The called routine would naturally have to know what order the parameters are in and what type of variable each parameter is. If the called function needs to return a value, it should put it back into the cassette buffer at
location $033c. Since the value is not written until just before returning, you don’t have to worry about overwriting what is already there.

**An Example Is Worth Two Thousand Bytes**

In order to clear up any questions you may still have, I will present a practical example of creating an object file from an assembly language file. While I used PAL as my assembler, you should be able to figure out how to get your assembler to do some of the unusual things necessary to create an object file. Unfortunately, the current version of SYMASH, the PAL-compatible assembler, will not be able to assemble my example because it requires assembling to disk (since the code is assembled to location $0000).

Listing 2 is a Doodle program written in C. It requires four external routines, which it will get from Listing 1, the assembly language portion. You will have to compile the C portion, assemble the assembly language portion, and then link them together with the linker. You will then have an executable program that will let you draw on the hires screen with the UKM diamond. The +, -, and / keys set the drawing mode to on, off, and flip respectively. RUN/STOP restores the normal text screen and exits the program. The program doesn’t do any boundary checking – so don’t try to draw off the screen or you may destroy something vital!

Listing 1 provides four functions: **Clear**, **Plot**, **FastKeys**, and **SlowKeys**. **Clear** fills any blank area of memory with any size with any byte. **Plot** allows you to manipulate any pixel on the graphic screen. **FastKeys** sets up an interrupt routine to speed up the keyboard repeat, and **SlowKeys** turns it off again.

Line 5 in listing 1 opens the object file to which it will write the object file. I am following a convention (which I suspect the author of Power C followed) of suffixing object files created from C source with a .O and object files created from Assembly source with .OBJ.

Lines 10 and 20 “fix” PAL so it writes the object file correctly for our purposes. Normally a machine language file begins with the object code origin address so that the kernal LOAD routine knows where to place the routine when you load it. The linker does not require that address and, in fact, gets confused by it. The pokes in line 20 replace the two JMP $FFD2 instructions that write the address to the file with do-nothing BIT $FFD2 instructions. If you have another assembler you will have to find a way to get around this problem. You may have to write a little program to strip the first two bytes off the object file after creating it.

Line 30 invokes PAL, and line 40 tells it to assemble to the file opened in line 10. In line 50 I tell the assembler to start assembling to location $0000 (minus two for the length word). I then define the filler label xxxxxxx in line 60. I use this label in references to external entities since the assembler requires something. The linker will fill in the correct address. Line 120 sets up the jump to the setup routine in case this object file is the first one to be linked in.

Lines 100, 7010, 8020, and 9020 set up the length word for each of the sections. In line 100 it is just a matter of putting the end of the code section since the code starts at $0000. The length in line 7010 is calculated by taking the number of bytes defined in the relocate section, dividing by two (since the length is expressed in words instead of bytes), and subtracting one (to skip the length word).

Calculating the length in the global and external sections is a little different. Here I use a label as if it were a variable, adding one for each entry, using PAL’s left-arrow temporary assignment operator. Since calculating labels happens in the first pass and the code is written the second pass, it doesn’t matter that the lines that increment the label (lines 8040, 8090, 8140, etc.) appear after the line putting the word in the file (line 8020 or 9020).

The **Clear** routine is in lines 160-390 and the global entry at lines 8090-8120 open this routine to allow access by other functions. Likewise, **Plot** in lines 500-1010 is opened by lines 8140-8170 as are the **FastKeys** and **SlowKeys** routines by the entries at lines 8190-8220 and 8240-8270 respectively.

Notice the global entry at lines 8290-8320 and the two external entries at lines 9280-9360 for the **irq%** routine. Sometimes you may need to access a local routine or variable in a more specialized way than just by absolute addressing. Lines 1120 and 1140 need to access the local routine **irqkeys** by immediate addressing. The relocate section, however, only relocates absolute addressing instructions. In order to get to work I had to treat the **irqkeys** routine as an external routine. This shows that local routines can be treated as external routines if necessary. Also, I chose to add two % symbols to the name to ensure that it doesn’t interfere if you happen to define another routine named irq somewhere else.

In line 970 I am storing a value back into location $033c. This is to provide a return value so that the calling routine can check the new state of the pixel after the **Plot** routine is called.

**In Conclusion...**

I would like to thank Brian Hilchie for a powerful compiler that has raised the productivity value of the Commodore 64 by several notches. Thanks also for an elegant and straightforward object format. But why didn’t he include this information in the documentation – to allow someone to make some money writing articles about it? I would suggest to Brian that, given the nature of C, machine language, and his specific implementation, it should not have been hard for him to include a *ASM and *ENDM set of compiler directives to allow inline assembly language. This would have made an attractive compiler virtually irresistible. I would recommend him adding it to a future update. After all, compared to writing a compiler, adding a simple assembler should be peanuts. He may be able to use PAL or SYMASH as a skeleton. If anyone could give me Brian Hilchie’s address, I would like to be able to write to him myself.

Those of you who want to take these ideas farther might want to tackle writing an assembler that would assemble source into object files of the type linkable by the C linker. You would probably need to add some pseudo ops like .GLOBAL, .EXTERN, and .FUNC.

If you want to discuss specifics for such an assembler, or have any questions, problems, corrections or criticisms, I would love to hear from you. I can be reached at the following address:

David Godshall
137 Wagner
Elkhart, IN 46516

Fido-Mail or Net-Mail can be sent to me at node 11/205 - <G>oshen <T>owne <C>rier.
### Table 1: Composition of Object Files

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Length 6502 instructions</td>
</tr>
<tr>
<td>Relocate</td>
<td>Length Address</td>
</tr>
<tr>
<td>Global</td>
<td>Length Global Entry</td>
</tr>
<tr>
<td>External</td>
<td>Length External Entry</td>
</tr>
</tbody>
</table>

#### Composition of Global Entries:
- **Name**: Name of the entry
- **Address**: Address of the entry
- **Mode**: 0 = Address is absolute, 1 = Address is local

#### Composition of External Entries:
- **Name**: Name of the entry
- **Offset**: Offset to add
- **Address**: Address x 4

#### Listing 1: GRPLOT.PAL

```
# HC 5 open 2,8,2,'@0:grplot.obj,s,w'
# FM 10 pal = peek(701) + 256*peek(702)
# DB 20 poke pal + 1759,44: poke pal + 1764,44
# PO 30 sys 700
# JD 40 .opt 02
# OH 50 == -2
# MF 60 = 0
# PM 89 ;
# OI 97 ;
# CM 98 ;
# AJ 99 ;
# MA 100 .word crelc
# EO 110 ;
# FB 120 cstart jmp xxxx
# IP 130 ;
# KD 131 ;
# DO 132 ;
# NH 133 ;
# KA 134 ;
# DP 135 ;
# EG 136 ;
# FP 137 ;
# CI 138 ;
# BA 139 ;
# FD 140 address .word $e000
# MA 150 ;
# JM 160 clear =

# DJ 170 lda $033c ;<addr
# CB 180 rloc sta laddr
# CI 190 sta $22
# MK 200 lda $033d ;>$addr
# JB 210 rloc sta laddr + 1
# DK 220 sta $23
# JO 230 lda $0340 ;byte
# DH 240 ldy #$00
# JE 250 ldx $033f ;<length
# GF 260 beq floop2
# DJ 270 floop1 sta ($22),y ;fill a
# CN 280 dey ;page
# NF 290 bne floop1
# KM 300 inc $23 ;fill many
# CJ 310 dex ;pages
# LH 320 bne floop1
# FJ 330 ldy #0
# JL 340 floop2 cpy $033e ;<length
# FE 350 beq fexit
# IF 360 sta ($22),y ;fill part
# ML 370 iny ;of a page
# JL 380 bne floop2
# LM 390 fexit rts
# FA 399 ;
# NF 400 grrows .word 0, 320, 640, 960,1280
# AM 410 .word 1600,1920,2240,2560,2880
# EL 420 .word 3200,3520,3840,4160,4480
# BN 430 .word 4800,5120,5440,5760,6080
```

---

*The Transactor* 39 March 1988: Volume 8, Issue 05
Listing 2: DOODLE.C
/*
doodle.c
*/

by David Godshall

char PlotType;

main()
{
    char *Pointer, Key, Store1, Store2, Store3;
    unsigned int Loop, X, Y;
    highmem (0xCC00);

    Pointer = 0xDD00;
    *Pointer = (Store1 = *Pointer) & 255;
    Pointer = 0xDD11;
    *Pointer = (Store2 = *Pointer) | 32;
    Pointer = 0xDD18;
    Store3 = *Pointer;
    *Pointer = 0x38;

    Pointer = 0x028a;
    *Pointer = 128;
    FastKeys();

    Clear (0xCC00, 1000, 93);
    Clear (0xEO00, 8000, 0);
    PlotType = 1;
    X = 160;
    Y = 100;
    Plot (X, Y);

    while ((Key = waitkey()) != 3)
    {
        switch (Key)
        {
            case 'I':
                case 'i':
                    Plot (X, Y);
                    /* Allow user to draw lines */
                    break;
            case 'M':
                Plot (X, Y);
                /* diamond, -, +, and / set */
                break;
            case 'N':
                Plot (X, Y);
                /* clear, set, or flip mode */
                break;
            case 'P':
                Plot (X, Y);
                /* respectively, STOP exits */
                break;
            case 'S':
                Plot (X, Y);
                break;
            case 'T':
                Plot (X, Y);
                break;
            case 'E':
                Plot (X, Y);
                break;
            case 'F':
                Plot (X, Y);
                break;
            case 'G':
                Plot (X, Y);
                break;
            case 'A':
                Plot (X, Y);
                break;
            case 'B':
                Plot (X, Y);
                break;
            case 'C':
                Plot (X, Y);
                break;
            case 'D':
                Plot (X, Y);
                break;
            case 'E':
                Plot (X, Y);
                break;
            case 'F':
                Plot (X, Y);
                break;
            case 'G':
                Plot (X, Y);
                break;
            case 'H':
                Plot (X, Y);
                break;
            case 'I':
                Plot (X, Y);
                break;
            case 'J':
                Plot (X, Y);
                break;
            case 'K':
                Plot (X, Y);
                break;
            case 'L':
                Plot (X, Y);
                break;
            case 'M':
                Plot (X, Y);
                break;
            case 'N':
                Plot (X, Y);
                break;
            case 'O':
                Plot (X, Y);
                break;
            case 'P':
                Plot (X, Y);
                break;
            case 'Q':
                Plot (X, Y);
                break;
            case 'R':
                Plot (X, Y);
                break;
            case 'S':
                Plot (X, Y);
                break;
            case 'T':
                Plot (X, Y);
                break;
            case 'U':
                Plot (X, Y);
                break;
            case 'V':
                Plot (X, Y);
                break;
            case 'W':
                Plot (X, Y);
                break;
            case 'X':
                Plot (X, Y);
                break;
            case 'Y':
                Plot (X, Y);
                break;
            case 'Z':
                Plot (X, Y);
                break;
        }
    }

    SlowKeys();

    Pointer = 0xDD00;
    *Pointer = Store1;
    Pointer = 0xDD11;
    *Pointer = Store2;
    Pointer = 0xDD18;
    *Pointer = Store3;
}

#define GETIN 0xFFE4

char a, x, y;

#define numkeys = 198;

/* Waits for user to press a key */

int waitkey()
{
    while (*numkeys == 0)
    {
        sys (GETIN, &a, &x, &y);
        return a;
    }
}