

C and Assembly: Clarifying the Link

Not all static variables are created equal...

by Larry Gaynier

Larry Gaynier has more than 15 years of experience in software design, functional specification and documentation in a variety of mainframe environments. In this article, he builds on the description of C object files given by David Godshall in Volume 8, Issue 5 and gives explicit detail of how the "C Power/Power C" compiler handles static variables. Some of this material was also discussed by Adrian Pepper in Volume 9, Issue 1.

I want to clarify some items presented in a recent *Transactor* article (*The Link Between C and Assembly* by David Godshall, *Transactor*, Volume 8, Issue 5). In my remarks, I assume "Power C" from Spinnaker is identical to "C Power" from Pro-Line. I have the "C Power" compilers for the C64 and the C128.

Section 5: the static section

The article described four sections that make up a C object file. However, there is a fifth section for static variables. The two NULLs, thought to terminate an object file, actually specify the length of the static section. In the example given, the length is zero, meaning there are no static variables.

An important characteristic of static variables is that they are always initialized to a known value at program start-up. The default initial value is zero unless an explicit value is included in the declaration for a static variable.

The static section of a C Power object file only contains static variables that rely on the default initialization to zero. Static variables that are explicitly declared with initial values, are handled in a different manner by the compiler. Unfortunately, this leads to inconsistent behavior with C Power programs as we shall see later.

Each entry in the static section consists of a null-terminated name string followed by a word giving the size in bytes.

Consider the following example:

```
int fun()
{
    char c;
    static int iarray[5];
    c = 1;
}
```

iarray is declared to be a static array of five integers. When compiled using C64 C Power, the following object file is produced.

hex dump of file fun.o

```
0000 1d 00 4c 4c 4c 85 fb a9  ..111...
0008 01 a2 00 a0 00 20 20 20  ... .
0010 a2 01 a0 00 86 2b a9 01  .. ..+..
0018 a2 00 a0 00 4c 4c 4c 00  .. .111.
0020 00 01 00 46 55 4e 00 01  ...fun..
0028 03 00 03 00 43 24 53 54  ....c$st
0030 41 52 54 00 00 00 00 00  art.....
0038 43 24 31 30 35 00 00 00  c$105...
0040 0b 00 43 24 31 30 36 00  ..c$106.
0048 00 00 1a 00 01 00 cd 29  .....M)
0050 49 41 52 52 41 59 00 05  iarray.H
0058 00
```

The sections in the object file can be easily identified based on their location relative to the beginning of the file.

Section	Location	Length	Contents
-----	-----	-----	-----
code	\$0000	\$1D	
relocate	\$001F	0	
global	\$0021	1	fun
external	\$002A	3	c\$start,c\$105, c\$106

The static variable section begins at location \$004C showing a length of one. Next, comes the only static entry - *iarray*. Its size is declared to be ten bytes. The first two bytes in the name string are generated by the compiler. (I am not sure about their significance. They appear to be random identifiers assigned by the compiler).

As you can see, no data space has actually been allocated for *iarray*. This happens at run-time similar to automatic variables. During the linking process, the static variable entries are collected into a contiguous data area to be located immediately above the program in memory. When linking is complete, the executable program knows the starting address and size of the static data area. At run-time, the program performs a simple loop to zero each byte in the static data area, guaranteeing the static variables are initialized to zero.

Inconsistent behavior

If a static variable is explicitly declared with an initial value, appropriate memory is allocated and initialized in the code section after the JMP C\$START but before the regular executable code. It does not appear in the static section. As a result, initialization happens once when the program is loaded. If the static variable changes value during execution, the new value is remembered and becomes the initial value for the next execution of the program, unless the program is reloaded. Consider the following example, which includes explicit initialization.

```
int fun1()
{
    char c;
    static int iarray[5] = {1,2,3,4,5};
    c = 1;
}
```

iarray is declared to be a static array of five integers initialized to 1,2,3,4,5. When compiled using C64 C Power, the following object file is produced:

hex dump of file fun1.o

```
0000 27 00 4c 4c 4c 01 00 02  '...lll...
0008 00 03 00 04 00 05 00 85  ....
0010 fb a9 01 a2 00 a0 00 20  ....
0018 20 20 a2 01 a0 00 86 2b  ....+
0020 a9 01 a2 00 a0 00 4c 4c  ....ll
0028 4c 00 00 01 00 46 55 4e  l....fun
0030 31 00 01 0d 00 03 00 43  1.....c
0038 24 53 54 41 52 54 00 00  $start..
0040 00 00 00 43 24 31 30 35  ...c$105
0048 00 00 00 15 00 43 24 31  ....c$1
0050 30 36 00 00 00 24 00 00  06...$.
0058 00
```

The sections in the object file can be easily identified based on their location relative to the beginning of the file.

Section	Location	Length	Contents
code	\$0000	\$27	
relocate	\$0029	0	
global	\$002B	1	fun1
external	\$0035	3	c\$start,c\$105, c\$106
static	\$0057	0	

Two things worth noting in this example are: the code section is larger and the static section is empty. This result occurred because *iarray* was allocated and initialized by the compiler beginning at location \$0005.

My advice is to keep in mind the potential side effects when you explicitly initialize static variables as part of their declaration.

Parameter passing

I think the article contains an error in describing how parameters are passed during a function call. A character variable was claimed to be passed as one byte. Actually, a character variable is first converted to an integer when passed. This conversion is described in the book "The C Programming Language" by Kernighan and Ritchie. You may see other literature about the C language refer to the conversion as 'widening' or 'promoting.' Any actual arguments of type *float* are converted to *double* before the function call; any of type *char* or *short* are converted to *int*. The C Power compiler only widens character variables because the data type sizes are limited: *char* is one byte; *short*, *int*, *long*, *unsigned* and *pointer* are two bytes; *float* and *double* are five bytes. Consider the following example:

```
callfred()
{
    char age, *name;
    float weight;
    int height;
    fred(age,name,weight,height);
}
```

This function calls the function FRED that was described in the article. When compiled using C64 C Power, the following object file is produced.

hex dump of file callfred.o

```
0000 48 00 4c 4c 4c 85 fb a9  h.lll...
0008 05 a2 05 a0 00 20 20 20  ....
0010 a9 00 20 20 20 a6 2b a0  ....+
0018 00 8e 3c 03 8c 3d 03 a6  ..<..=..
0020 2c a4 2d 8e 3e 03 8c 3f  ,->...?
0028 03 a9 04 a2 00 a0 00 20  ....
0030 20 20 a6 2e a4 2f 8e 45  ..../.e
0038 03 8c 46 03 a9 0b 20 6c  ..f...
0040 6c a9 05 a2 05 a0 00 4c  ....l
```

```

0048 4c 4c 00 00 01 00 43 41 11....ca
0050 4c 4c 46 52 45 44 00 01 11fred..
0058 03 00 06 00 43 24 53 54 ....c$st
0060 41 52 54 00 00 00 00 00 art.....
0068 43 24 31 30 35 00 00 00 c$105...
0070 0b 00 43 24 31 31 30 32 ..c$1102
0078 00 00 00 10 00 43 24 31 .....c$1
0080 31 33 38 00 00 00 2d 00 138...-.
0088 46 52 45 44 00 00 00 3c fred...<
0090 00 43 24 31 30 36 00 00 .c$106..
0098 00 45 00 00 00 .e...

```

On disassembly, it will be seen that the following 6502 code is produced:

Code Size: 72(D) / 48(X)

```

0 relocation entries
;-----
1 external definitions
callfred      R 0003
6 external references.

```

*= \$0000

```

4c 4c 4c / 1800    jmp c$start
;-----
callfred

```

```

85 fb 00 / 0003    sta $fb
a9 05 00 / 0005    lda #$05
a2 05 00 / 0007    ldx #$05
a0 00 00 / 0009    ldy #$00
20 20 20 / 000b    jsr c$105
a9 00 00 / 000e    lda #$00
20 20 20 / 0010    jsr c$1102
a6 2b 00 / 0013    ldx $2b
a0 00 00 / 0015    ldy #$00
8e 3c 03 / 0017    stx $033c
8c 3d 03 / 001a    sty $033d
a6 2c 00 / 001d    ldx $2c
a4 2d 00 / 001f    ldy $2d
8e 3e 03 / 0021    stx $033e
8c 3f 03 / 0024    sty $033f
a9 04 00 / 0027    lda #$04
a2 00 00 / 0029    ldx #$00
a0 00 00 / 002b    ldy #$00
20 20 20 / 002d    jsr c$1138
a6 2e 00 / 0030    ldx $2e
a4 2f 00 / 0032    ldy $2f
8e 45 03 / 0034    stx $0345
8c 46 03 / 0037    sty $0346
a9 0b 00 / 003a    lda #$0b
20 6c 6c / 003c    jsr fred
a9 05 00 / 003f    lda #$05
a2 05 00 / 0041    ldx #$05
a0 00 00 / 0043    ldy #$00
4c 4c 4c / 0045    jmp c$106

```


As the example shows, the accumulator is loaded with eleven just before the JSR to FRED. These eleven bytes of parameters are passed to FRED in memory starting at \$033c:

\$033c - Age (two bytes, zero high byte)
 \$033e - Name (two bytes, pointer)
 \$0340 - Weight (five bytes, FP representation)
 \$0345 - Height (two bytes)

Once inside the function FRED, the upper byte of Age will be loaded to zero page storage but it will never be used. Widening of parameters has been generally recognized as an inefficiency of the C language. The new ANSI C standard, when adopted, will add function prototyping to the C language which will make parameter widening unnecessary. I am curious to see if the "C Power" compilers will be updated to match the ANSI C standard.

Wrapup

Overall, the C Power 128 compiler exhibits identical behavior as the C64 version except that parameters are passed in memory starting at \$0400 in bank 1 and different zero page locations are used during function execution.

I hope you find this information useful to better understand the C Power compilers and to avoid some pitfalls. 

BIG BLUE READER 128/64 COMMODORE <=> IBM PC File Transfer Utility

Big Blue Reader 128/64 is ideal for those who use IBM PC compatible MS-DOS computers at work and have the Commodore 128 or 64 at home.

Big Blue Reader 128/64 is not an IBM PC emulator, but rather it is a quick and easy to use file transfer program designed to transfer word processing, text and ASCII files between two entirely different disk formats; Commodore and IBM MS-DOS. *Both C128 and C64 applications are on the same disk and requires either the 1571 and/or 1581 disk drive.* (Transfer 160K-360K 5.25 inch & 720K 3.5 inch MS-DOS disk files.)

Big Blue Reader 128 supports: C128 CP/M files, 17xx RAM exp. 40 and 80 column modes.

Big Blue Reader 64 Version 2 is 1571 and 1581 compatible and is available separately for only \$29.95!

BIG BLUE READER 128/64 only \$44.95

Order by check, money order, or C.O.D.
 No credit card orders please. Foreign orders add \$4
 BBR 128/64 available to current BBR users for \$18 plus your original disk.
 Free shipping and handling. CALL or WRITE for more information.



To order Call or Write:
SOGWAP Software

115 Belmont Rd
 Decatur, IN 46733
 Ph (219) 724-3900