

OVERVIEW

Function pointers are one of the many difficult features of the C programming language. Due to the unique requirements of a C compiler for the 8051 architecture, function pointers and reentrant functions have even greater challenges to surmount. This is primarily due to the way function arguments are passed.

Typically, (for most every chip other than the 8051) function arguments are passed on the stack using the push and pop assembly instructions. Since the 8051 has a size limited stack (only 128 bytes and as low as 64 bytes on some devices), function arguments must be passed using a different technique.

When Intel introduced the PL/M-51 compiler for the 8051, they introduced the technique of storing arguments in fixed memory locations. When the linker was invoked, it built a call tree of the program, figured out which function arguments were mutually exclusive, and overlaid them. This was the beginning of the linker's **OVERLAY** directive.

Since PL/M-51 doesn't support function pointers, the issue of indirect function calls never came up. However, with C, problems abound. How does the linker "know" which memory to use for the indirect function's arguments? How do you add functions that are called indirectly into the call tree?

This document explains how to effectively use function pointers in your C51 programs. Several examples are used to illustrate the problems and solutions that are discussed. Specifically, the following topics are discussed.

- Casting a Constant Address to a Pointer
- Declaring Function Pointers
- Problems with Function Pointers in C51
- Using the **OVERLAY** Directive to Fix the Call Tree
- Pointers to Reentrant Functions

POINTERS TO FIXED ADDRESSES

You can easily type cast numeric addresses into function pointers. There are numerous reasons to do this. For example, you may need to reset the target and application without toggling the reset line to the CPU. You can use a function pointer to address 0000h to accomplish this.

You may use the type casting features of Standard C to cast 0x0000 into a pointer to a function at address 0. For example, when you compile the following line of C code...

```
((void (code *) (void)) 0x0000) ();
```

... the compiler generates the following:

```

; FUNCTION main (BEGIN)
; SOURCE LINE # 3
0000 120000      LCALL    00H
; SOURCE LINE # 4
0003 22         RET
; FUNCTION main (END)
    
```

This is exactly what we expected: LCALL 0.

Type casting a numeric constant into a function pointer is a tricky thing. The following description of the components of the above function call will help you understand how to better use them.

In the above function call, (void (*) (void)) is the data type: a pointer to a function that takes no arguments and that returns a **void**.

The 0x0000 is the address to cast. After the type cast, the function pointer points to address 0x0000. Note that we put parentheses around the data type and the 0x0000. This is unnecessary if we only want to cast 0x0000 to a function pointer. However, since we are going to invoke the function, these parentheses are required.

Casting a numeric constant to a pointer is not the same as calling a function through a pointer. To do that, we must specify an argument list. That is what the () at the end of the line does.

Note that all parentheses in this expression are required. Grouping and precedence are important.

The only difference between the above pointer and a pointer to a function with arguments is the data type and the argument list. For example, the following function call...

```
((long (code *) (int, int, int)) 0x8000) (1, 2, 3);
```

... invokes a function, at address 0x8000, that accepts three (3) **int** arguments and returns a **long**.

POINTERS TO FUNCTIONS WITH NO ARGUMENTS

Function pointers are variables that point to a function. The value of the variable is the address of the function. For example, the following function pointer declaration...

```
void (*function_ptr) (void);
```

...is a pointer called `function_ptr`. Use the following code to invoke the function that `function_ptr` points to.

```
(*function_ptr) ();
```

Since the function that `function_ptr` points to takes no arguments, none are passed. That's why the argument list is empty.

The address that `function_ptr` calls may be assigned when the variable is declared:

```
void (*function_ptr) (void) = another_function;
```

Or, it may be assigned during program execution, at run-time.

```
function_ptr = and_another_function;
```

It is important to note that you **must** assign an address to a function pointer. If you don't, the pointer may have a value of 0 (if you're lucky) or it may have some completely undetermined value, depending on how your data memory is used. If the pointer is uninitialized when you indirectly call a function through it, your program will probably crash.

To declare a pointer to a function with a return value you must specify the return type in the declaration. For example, the following declaration changes the above declaration to point to a function that returns a float.

```
float (*function_ptr) (void) = another_function;
```

Pretty simple stuff. Just remember where the parentheses go and it's easy.

POINTERS TO FUNCTIONS WITH ARGUMENTS

Pointers to functions with arguments are similar to pointers to functions with no arguments. For example:

```
void (*function_ptr) (int, long, char);
```

...is a pointer to a function that takes an **int**, a **long**, and a **char** as arguments. Use the following to invoke the function that `function_ptr` points to.

```
(*function_ptr) (12, 34L, 'A');
```

Note that function pointers may only point to functions with three (3) arguments or less. This is because the arguments of indirectly called functions must reside in registers. Refer to reentrant functions below for information about pointers to functions that use more than three arguments.

CAVEATS OF USING FUNCTION POINTERS

There are several caveats you must be aware of if you use function pointers in your C51 programs.

ARGUMENT LIST LIMITS

Arguments passed to functions through a function pointer must all fit into registers. At most three (3) arguments can automatically be passed in registers. Refer to the C51 User's Manual to find out the algorithm for fitting arguments into registers. Do not assume that just any three argument data types will fit.

Since C51 passes up to three arguments in registers, the memory space used for passing arguments is not an issue unless the function pointed to requires more arguments. If that is the case, you can merge the arguments into a structure and pass a pointer to the structure. If that is not acceptable, you may always use reentrant functions (see below).

CALL TREE PRESERVATION

The C51 tool chain does not push function parameters onto the stack (unless reentrant functions are used). Instead, function parameters and automatic variables (locals) are stored in registers or in fixed memory locations. This prevents functions from being reentrant. For example, if a function calls itself, it overwrites its own arguments and/or locals when it calls itself. The problem of reentrancy is addressed by the **reentrant** keyword (see below). Another side-effect of non-reentrancy is that function pointers can, and usually do, present implementation problems.

In order to preserve as much data space as possible, the linker performs call tree analysis to determine if some memory areas may be safely overlaid. For example, if your application consists of the main function, function a, function b, and function c; and if main calls a, b, and c; and if a, b, and c call no other functions (not even each other); then, the call tree for your application appears as follows:

```

MAIN
  +--> A
  +--> B
  +--> C
    
```

And, the memory used by A, B, and C may be safely overlaid.

The problem with function pointers comes when the call tree cannot be correctly constructed. The reason is that the linker cannot determine which function a function pointer references. There is no automatic way around this problem. However, there is a manual one, albeit a bit cumbersome.

Function Pointers in C51

APNT_129

The following two source files help illustrate the problem and make the solution easier to understand. The first source file, FPCALLER.C, contains a function that calls another function through a function pointer (fptr).

```
void func_caller (
    long (code *fptr) (unsigned int))
{
    unsigned char i;

    for (i = 0; i < 10; i++)
    {
        (*fptr) (i);
    }
}
```

The second source file, FPMAIN.C, contains the main C function as well as the function that is called indirectly by func_caller (defined above). Note that main calls func_caller and passes the address of func as an argument.

```
extern void func_caller (long (code *) (unsigned int));

int func (unsigned int count)
{
    long j;
    long k;

    k = 0;
    for (j = 0; j < count; j++)
    {
        k += j;
    }

    return (k);
}

void main (void)
{
    func_caller (func);
    while (1) ;
}
```

The above two source files compile with **no errors**. They also link with **no errors**. The following call tree is produced in the map file created by the linker.

| SEGMENT | DATA_GROUP | |
|--------------------------------|------------|--------|
| +--> CALLED SEGMENT | START | LENGTH |
| ----- | | |
| ?C_C51STARTUP | ----- | ----- |
| +--> ?PR?MAIN?FPMAIN | | |
| ?PR?MAIN?FPMAIN | ----- | ----- |
| +--> ?PR?_FUNC?FPMAIN | | |
| +--> ?PR?_FUNC_CALLER?FPCALLER | | |
| ?PR?_FUNC?FPMAIN | 0008H | 000AH |
| ?PR?_FUNC_CALLER?FPCALLER | 0008H | 0003H |

There is a lot of information that can be derived from the call tree even in this simple example.

Function Pointers in C51

APNT_129

The ?C_C51STARTUP segment calls the MAIN C function this is the ?PR?MAIN?FPMAIN segment. This components of this segment name may be decoded as: PR is the something in the PProgram memory, MAIN is the name of the function, and FPMAIN is the name of the source file where the function is defined.

The MAIN function calls FUNC and FUNC_CALLER (according to the call tree). Note that this is not correct. MAIN never calls FUNC. But, it does pass the address of FUNC to FUNC_CALLER. Also note that according to the call tree, FUNC_CALLER does not call FUNC. This is because it is called indirectly through a function pointer.

The FUNC function in FPMAIN uses 000Ah bytes of DATA which starts at 0008h. FUNC_CALLER in FPCALLER uses 0003h bytes of DATA which also starts at 0008h. This is important!

FUNC_CALLER uses memory starting at 0008h and FUNC also uses memory starting at 0008h. Since FUNC_CALLER invokes FUNC and since both functions use the same memory area we have a problem. When FUNC gets called (by FUNC_CALLER) it trashes the memory used by FUNC_CALLER. How did this happen? Don't the Keil C51 Compiler and Linker work?

The cause of the problem here is the function pointer. Whenever you use function pointers, you will almost always have these types of problems. Fortunately, they are easy to fix. The OVERLAY linker directive lets you specify how functions are linked together in the call tree.

To correct the call tree shown above, the call to FUNC must be removed from the MAIN function and a call to FUNC must be inserted under the FUNC_CALLER function. The following OVERLAY command does just that.

```
OVERLAY (?PR?MAIN?FPMAIN ~ ?PR?_FUNC?FPMAIN,
        ?PR?_FUNC_CALLER?FPCALLER ! ?PR?_FUNC?FPMAIN)
```

To remove or insert references to the call tree, specify the caller first and the callee second. Tilde (~) removes a reference or call and the exclamation point (!) adds a reference or call. For example, ?PR?MAIN?FPMAIN ~ ?PR?_FUNC?FPMAIN removes the call from MAIN to FUNC.

After the linker command is adjusted to include the OVERLAY directive to correct the call tree, the map file appears as follows:

| SEGMENT | DATA_GROUP | START | LENGTH |
|--------------------------------|------------|-------|--------|
| +--> CALLED SEGMENT | | | |
| ----- | | | |
| ?C_C51STARTUP | | ----- | ----- |
| +--> ?PR?MAIN?FPMAIN | | | |
| ?PR?MAIN?FPMAIN | | ----- | ----- |
| +--> ?PR?_FUNC_CALLER?FPCALLER | | | |
| ?PR?_FUNC_CALLER?FPCALLER | | 0008H | 0003H |
| +--> ?PR?_FUNC?FPMAIN | | | |
| ?PR?_FUNC?FPMAIN | | 000BH | 000AH |

And, the call tree is now correct and the variables for FUNC and FUNC_CALLER are now in separate spaces (and are not overlaid).

TABLES OF FUNCTION POINTERS

The following is a typical function pointer table definition:

```
long (code *fp_tab []) (void) =
{ func1, func2, func3 };
```

If your main C function calls functions through fp_tab, the link map appears as follows:

| SEGMENT | DATA_GROUP | START | LENGTH |
|------------------------|------------|-------|--------|
| +--> CALLED SEGMENT | | | |
| ----- | | | |
| ?C_C51STARTUP | | ----- | ----- |
| +--> ?PR?MAIN?FPT_MAIN | | | |
| +--> ?C_INITSEG | | | |
| ?PR?MAIN?FPT_MAIN | | 0008H | 0001H |
| ?C_INITSEG | | ----- | ----- |
| +--> ?PR?FUNC1?FP_TAB | | | |
| +--> ?PR?FUNC2?FP_TAB | | | |
| +--> ?PR?FUNC3?FP_TAB | | | |
| ?PR?FUNC1?FP_TAB | | 0008H | 0008H |
| ?PR?FUNC2?FP_TAB | | 0008H | 0008H |
| ?PR?FUNC3?FP_TAB | | 0008H | 0008H |

The three functions called through the table, func1, func2, and func3, appear as though they are called by ?C_INITSEG. However, this is incorrect. ?C_INITSEG is the routine that initializes the variables in your program. These functions are referenced in the initialization code because the function pointer table is initialized with the addresses of these functions.

Note that the starting area for variables used by the main C function as well as func1, func2, and func3 all start at 0008h. This won't work because the main C function calls func1, func2, and func3 (through the function pointer table). And, the variables used in func1, et. al. overwrite those used in main.

The C51 Compiler and BL51 Linker work in combination to make overlaying the variable space of function easy when you use tables of function pointers. However, you must declare the pointer tables appropriately. If you do this, you can avoid using the OVERLAY directive. The following is a function pointer table definition that C51 and BL51 can handle automatically:

```
code long (code *fp_tab []) (void) =
{ func1, func2, func3 };
```

Note that the only difference is storing the table in CODE space.

Now, the link map appears as follows:

| SEGMENT | DATA_GROUP | |
|------------------------|------------|--------|
| +--> CALLED SEGMENT | START | LENGTH |
| ?C_C51STARTUP | ----- | ----- |
| +--> ?PR?MAIN?FPT_MAIN | | |
| ?PR?MAIN?FPT_MAIN | 0008H | 0001H |
| +--> ?CO?FP_TAB | | |
| ?CO?FP_TAB | ----- | ----- |
| +--> ?PR?FUNC1?FP_TAB | | |
| +--> ?PR?FUNC2?FP_TAB | | |
| +--> ?PR?FUNC3?FP_TAB | | |
| ?PR?FUNC1?FP_TAB | 0009H | 0008H |
| ?PR?FUNC2?FP_TAB | 0009H | 0008H |
| ?PR?FUNC3?FP_TAB | 0009H | 0008H |

Now there is no reference from the initialization code to func1, func2, and func3. Instead, there is a reference from main to the constant segment FP_TAB. This is the function pointer table. Since the function pointer table references func1, func2, and func3, the call tree is correct.

As long as the function pointer table is located in a separate source file, C51 and BL51 make all the correct links in the call tree for you.

FUNCTION POINTER TIPS AND TRICKS

There are some function pointer tricks you can use to make things easier.

USE MEMORY-SPECIFIC POINTERS

Change the function pointer from a generic pointer to a memory-specific pointer. This saves one (1) byte for each pointer. The examples used so far used declared generic function pointers. Since functions only reside in CODE memory (on the 8051), one byte may be saved by declaring the functions pointer as a code pointer. For example:

```
void (code *function_ptr) (void) = another_function;
```

If you choose to include the code keyword in your function pointer declarations, make sure you use it everywhere. If you declare a function that accepts a generic 3-byte function pointer and the pass it a memory-specific, 2-byte function pointer, bad things will happen!

REENTRANT FUNCTIONS AND POINTERS

Keil C51 offers the **reentrant** keyword for functions that are reentrant. Reentrant functions expect arguments to be passed on a simulated stack. The stack is maintained in **IDATA** for small memory model, **PDATA** for compact memory model, or **XDATA** for large memory model. You must initialize the reentrant stack pointer in `STARTUP.A51` if you use reentrant functions. Refer to the following excerpt from the startup code.

```

;-----
; Reentrant Stack Initialization
;
; The following EQU statements define the stack pointer for reentrant
; functions and initialized it:
;
; Stack Space for reentrant functions in the SMALL model.
IBPSTACK EQU 0 ; set to 1 if small reentrant is used.
IBPSTACKTOP EQU 0FFH+1 ; set top of stack to highest location+1.
;
; Stack Space for reentrant functions in the LARGE model.
XBPSTACK EQU 0 ; set to 1 if large reentrant is used.
XBPSTACKTOP EQU 0FFFFH+1 ; set top of stack to highest location+1.
;
; Stack Space for reentrant functions in the COMPACT model.
PBPSTACK EQU 0 ; set to 1 if compact reentrant is used.
PBPSTACKTOP EQU 0FFFFH+1 ; set top of stack to highest location+1.
;-----
    
```

You must set the equate for which memory model stack you use and set the top of the stack. The reentrant stack pointer decreases (moves down) as items are pushed onto the stack. A neat trick for conserving internal **data** memory is to place all reentrant functions in a separate memory model like large or compact.

To declare a reentrant function, use the **reentrant** keyword.

```

void reentrant_func (long arg1, long arg2, long arg3) reentrant
{
}
    
```

To declare a large model reentrant function, use the **large** and **reentrant** keywords.

```

void reentrant_func (long arg1, long arg2, long arg3) large reentrant
{
}
    
```

To declare a function pointer to a reentrant function, you must also use the **reentrant** keyword.

```

void (*rfunc_ptr) (long, long, long) reentrant = reentrant_func;
    
```

There is not much difference between declaring reentrant function pointers and non-reentrant function pointers.

When using pointers to reentrant functions, more code is generated because arguments must be pushed onto the simulated stack. However, no special linker controls are required and you need not mess with the **OVERLAY** directive.

If you pass more than three (3) arguments to a function that you call indirectly, reentrant function pointers are required.

CONCLUSION

Function pointers are useful and not too terribly difficult to use if you pay attention to the linker call tree and make sure to use the **OVERLAY** directive to correct any inconsistencies.

Copyright © 1999 Keil Software, Inc. All rights reserved.

In the USA:
Keil Software, Inc.
16990 Dallas Parkway, Suite 120
Dallas, TX 75248-1903
USA

Sales: 800-348-8051
Phone: 972-735-8052
FAX: 972-735-8055

E-mail: sales.us@keil.com
support.us@keil.com

Internet: <http://www.keil.com/>

In Europe:
Keil Elektronik GmbH
Bretonischer Ring 15
D-85630 Grasbrunn b. Munchen
Germany

Phone: (49) (089) 45 60 40 - 0
FAX: (49) (089) 46 81 62

E-mail: sales.intl@keil.com
support.intl@keil.com