C is the language of choice for any type of systems programming (including embedded systems), because it allows the programmer to gain easy access to the hardware. Unfortunately, the built-in support for that access isn't as great as what C's reputation might imply.

Depending on the environment, programmers may need to directly manipulate memory, registers, processor status flags, I/O ports, or interrupt vectors and handlers. Pointers do allow direct access to memory, but all other hardware resources must be accessed with assembly language, either through a separately written and linked function, in-line assembly (if your compiler allows it), or very nasty tricks like executable strings.

Fortunately, because C is so often used for systems programming, most compiler vendors include in their standard library package routines for manipulating ports and restricted aspects of interrupts. Mainstream compilers, though, seldom include a means of accessing processor flags and register contents, or completely general support for interrupt use. Thus, if you need to manipulate only memory and ports, you may be able to avoid writing any assembly in your project by becoming familiar with the support built into your compiler's library. Otherwise you will need to resort to the techniques developed here.

ACCESSING MEMORY

On a machine with a strictly linear address space (like the 68000 series), manipulating memory is very straightforward. C's pointers are special variables which are assumed to hold a memory address. The asterisk (*) operator fetches or writes a value to the memory address in the pointer. Thus to prepare to write a byte to location 0x05 (perhaps to set the I/O byte on an old CP/M system):

```c
char *loc;
loc = (char *) 0x05;
```

(The cast is more than just good form. On word-oriented and some segmented architectures, byte addresses are not always represented as simple integers. If you don't cast the integer, you may not get a pointer that works like you expect.)
To write a seven into this location:

*loc = 7;

To copy the contents of location five into a variable that may be manipulated by a C program:

char workspace;

workspace = *loc;

It is important that the type of the pointer agree with the type of the data element you intend to manipulate, otherwise you may effect more or less data than you intended.

For example:

char *loc;
loc = (char *) 7;
*loc = 276;

actually puts 20 in location 7, because a character pointer only transfers one byte.

Similarly:

char *wideloc;
wideloc = (int *) 7;
*wideloc = 276;

may compile and run (perhaps with a warning), but will still only put 20 in location 7.

Big endian versus little endian conventions may also complicate memory accesses. For example, if you create a data table in assembly that looks like this:

```
ORG 100H
DB 07H, 09H, 13H, 42H, 47H
```

and then execute this code:

```
int entry;
int *secondint;
secondint = (int *) 0x102;
entry = *secondint;
printf("%x", entry);
```

You may be surprised with this result:

4213
Some machines assume integers are stored smallest byte first, others assume they are stored largest byte first.

Segmented architectures have their own quirky requirements. To exploit the efficiency benefits of the segmented architectures, programs for these machines are usually compiled in a mode that uses 16 bit pointers — i.e., pointers that are relative to some default base or segment register. If you want to access some data item that can be reached by a relative offset from the default register, the process is as simple as in a linear address (i.e. if you are staying within the present 64K data segment). But, if you need to access an address outside of this segment (say the video RAM page), things are a little more complicated. You must use a special pointer that references off of a different base register or changes and then restores the default register. Modern 8086 compilers offer built-in support for this latter operation in far pointers. However, the use of these special pointers is not always obvious. For example, one could reasonably expect to write to the second line of a PC clone’s video RAM (physical location 0xB0140) with:

```c
char far *video;
video = (char far *) 0xB0140L;
*video = 'X';
```

Unfortunately, this crashes the system. The problem is that far pointers aren’t simple integer representations of memory addresses. Instead they are segments (all but the least four bits of the address) and offsets (only the least 16 bits of the address). The code to correctly initialize the far pointer is:

```c
char far *video;
/* concatenate OxBO00 and 0140 */
video = (char far *) 0xB0000140L;
```

Pointers, together with bit operators, are all you need to conveniently handle all low-level hardware operations on a machine with memory-mapped I/O. If, for example, bit two of location FFF00 was the keyboard ready status bit on a machine with a simple linear memory structure, you could wait on a character directly in C with:

```c
char nextchar, *kbdst;
kbdst = 0xFF00;
while (!((nextchar=kbdst) & 0x2));
```

That’s how C earned it’s reputation for being a hardware language.

**INCORPORATING ASSEMBLY LANGUAGE**

C has no built-in understanding of non built-in operators for any address space other than memory. Thus if your machine uses port-mapped I/O, you must use the machine’s native operators to manipulate the ports. The same applies to special resources like special registers (e.g. a built-in UART) and CPU status flags.
Even if your compiler vendor's library includes routines to manipulate port space, special registers, and to disable and enable interrupts, you may still find a need to incorporate your own assembly language. I'll explain three different methods: modifying compiler output, using in-line sequences, and executable strings.

MODIFYING COMPILER OUTPUT

The most complicated part of writing a C library routine in assembly is getting the calling interface right. Why not let the compiler do it for you? If you compile to assembly (instead of to object code), a function with the same number and type of parameters as you need in your assembly language function, you can edit the compiler's output to produce the finished function. For example, if I needed to write a routine to get a byte from an arbitrary port, I would compile the function:

```c
extern int mark;

char inp(address)
int address;
{
    mark = address + 3;
    return (mark + 7);
}
```

I then edit the assembly listing to remove the external declaration and replace the dummy calculations with a port operation:

```assembly
/* compiler generated */
; Line 6
;    address = 4
; Line 7
    mov     ax,WORD PTR [bp+4] ;address
    add     ax,3
    mov     mark,ax
; Line 8
    add     ax,7
/* replacement */
    push    dx ;save dx in case
    mov     dx,WORD PTR [bp+4] ;get port # in dx
    in      ax,dx ;get byte in ax
    pop     dx ;restore
```

Referencing an external in the dummy code and performing some simple calculation on the parameter and return values "mark" the lines that access the parameter and "mark" the registers that are available for your use.

One advantage of using the compiler generated code: stack probes and other built-in protection mechanisms can be easily incorporated into your module.
IN-LINE ASSEMBLY

In the "good old days", Assembly language could be passed directly to a compiler with the pre-processor escape

```asm
... 
#endasm
```

Because this capability doesn't belong in the pre-processor, ANSI conforming compilers won't support it. Many will, however, support the same functionality through pragmas.

In either form, this capability is best reserved for single operations that don't change memory. For example:

To output a constant to port 7:

```asm
    OUT 7,0CH
#endasm
```

To enable interrupts

```asm
    EI
#endasm
```

To capture status flags in an external variable

```asm
    PUSH AX  ;save in case the compiler is using it
    LAHF
    MOV stbyte,AH
    POP AX  ;restore ...
#endasm
```

If you change a register, unless you are very knowledgable about how your compiler generates code, it's probably best to save registers as in this last example. In general, though, if you are going to use registers and modify memory, it's probably best to write the entire module in assembly, because it's very hard to predict how your use of memory and registers will interact with the compiler's.

EXECUTABLE STRINGS

In very simple environments, it is sometimes practical to hand-assembly a few instructions and insert them in a string. It is usually not too hard to convince a simple-minded compiler that a pointer to the string represents a function.
For example, to disable interrupts on a Z-80 we could code:

```c
void (*cheapf)();
char * asmcode;
/* D1 ; OxF3 = 363 ; RET = 0xC9 = 311 */
asmcode = "\363\311";
cheapf = asmcode;
(*cheapf)(); /* call to the code via the pointer */
```

Your compiler should at least complain about this abuse, but may produce executable code anyway. This technique will NOT work if relocatable objects are manipulated by the string code. I can’t in good conscience recommend that you ever use this technique, but here it is just in case.

**INTERRUPTS**

Of course you can manage and service interrupts by incorporating assembly language using any of the techniques discussed above. Some compilers (including Turbo C) include special keywords and pragmas that reduce the likelyhood you will have to resort to assembly. Of particular use are the ability to label a function as needing a “return from interrupt” exit, and library functions to enable, disable, and re-vector interrupts.

**SUMMARY**

C gives excellent direct control over memory and memory-mapped I/O. Other hardware resources are less directly accessible, but C compiler vendors tend to supply library functions that make it easy to manipulate all but the lowest-level hardware resources. In compilers targeted for the embedded system programmer, you should also find in-line assembly pragmas, special keywords, and machine specific library support. Even if these tools are missing, as long as you can make your compiler generate assembly, it’s fairly easy to add your functions to manipulate special resources.