Casting and Data Abstraction in C

Written by Kerry Crouse

When designing embedded systems, there exists the problem of accessing memory mapped registers. There are a number of methods which can be used to access memory mapped registers and setting up pointers to these locations is the most frequent method used to access these registers. Since I've been programming, I've come across several ways of specifying these registers: using the actual numeric address, using a variable containing the address, using a variable containing a pointer to the address, using a macro containing the address, using a macro containing a pointer to the address, and using a macro containing a dereferenced pointer to the address.

Numeric Address

This is simply the address of a register with an appropriate casting. For instance, a byte location at hexadecimal address 1000 would be cast as:

```
((unsigned char *) 0x1000)
```

This cast leaves us with a pointer to location 0x1000. To use this pointer, we must dereference the pointer using an additional '*' so we can actually assign data. The construct appears as:

```
*((unsigned char *) 0x1000) = 86;
```

The advantage of this approach is the visibility of the address and the speed of execution. Upon inspection we can see this is data.

Variable Storage

A variable may be used to hold the address. This approach ends up very similar to the Numeric Address approach. For example:

```
unsigned char *SerRcv = (unsigned char *) 0x1000;
*SerRcv = 86;
```

The advantages of the Variable storage approach is that the address can be modified at any time, the address can be changed in one place, and a symbolic name can be used for the location. The down side of this technique is that the variable location must be calculated for each access which begs the question: how often, if ever, will the address change at run time.

Macro Address

This is similar to the Numeric address in a macro wrapper. Using the example above, the code appears as:

```
#define SER_XMT_ADR 0x1000
```

An assignment appears as:

```
*((unsigned char *) SER_XMT_ADR) = 86;
```

Advantages are the address can be changed at one place at compile time and a symbolic name is used. The address, however, is used in a cumbersome way.

Macro Address Pointer

This is the same as the Numeric address with a macro wrapper. Using the example above, the code looks like this:

```
#define SER_XMT_P ((unsigned char *) 0x1000)
```

An assignment looks like this:

```
*SER_XMT_P = 86;
```

The advantages are that a symbolic name is used and the address can be changed in only one place at compile time.

Dereferenced Macro Address Pointer

This is almost the same as the Macro address pointer. Using the example above, the code looks like this:

```
#define SER_XMT (*((unsigned char *) 0x1000))
```

An assignment looks like this:

```
SER_XMT = 86;
```

The advantages are the address can be changed at one place at compile time, a representative symbolic name is used, and the location looks and behaves like a built-in register.

Using Dereferenced Macro Address Pointers

If the pointers are part of a larger I/O structure, the whole structure may be embedded into a macro representation of the I/O architecture.

```
#define IO_BASE 0x1000
#define SER_BASE (IO_BASE + 0x100)
#define SER_RECV (SER_BASE)
#define SER_XMIT (SER_BASE + 4)
#define SER_STAT (SER_BASE + 8)

#define PAR_BASE (IO_BASE + 0x200)
#define PAR_DATA (PAR_BASE)
#define PAR_STAT (PAR_BASE + 4)
```

Other Dereferenced Macro Address Pointers

Not only can base address types be used as address pointers, but structures can also be used. An example follows:

```
struct Serial
{
   char SerData_Stat
   char SerMdm
   char SerLn
};
struct Serial *Ser1 = (struct Serial *) 0x1000;
Ser1.SerData_Stat = 0x86
```

An example of a bitfield looks like this:

```
#define SER_STAT *((char *)(SER_BASE + 8))
#define SER_RCV_RDY ((SER_STAT & 0x80) != 0)
```

The serial status register is used to check the serial receiver bit. This check won't work if the bit resets after it is read.

Summary

The method of addressing a fixed memory location should be efficient from a performance standpoint, efficient from a speed perspective, easy for programmers to understand, and easy to change without breaking the code. All of the address calculations are done in a macro, therefore the performance is excellent. There are no variables involved, so the speed is fast. Due to the symbolic name, the function is easy for programmers to understand. Lastly, since the address calculations drop out of a macro, register and even board level locations can be changed easily. A macro using dereferenced cast pointer constants can provide all of these attributes.