How to Program an 8-bit Microcontroller Using C language

By: Richard Mann, Imagecraft

TRADITIONALLY, MOST 8-BIT EMBEDDED PROGRAMS HAVE BEEN WRITTEN IN ASSEMBLY LANGUAGE. HOWEVER, DUE TO A VARIETY OF REASONS, MOST NEW MICROCONTROLLERS INCLUDING THE 8-BIT ONES ARE NOW EXPECTED TO HAVE A C COMPILER AVAILABLE. COMPARED TO AN EQUIVALENT ASSEMBLY PROGRAM, A WELL-WRITTEN C PROGRAM IS TYPICALLY MORE READABLE AND MAINTAINABLE. PLUS, WITH SOME CARE AND SOME AMOUNT OF CHANGES, THE C PROGRAM MAY BE MOVED TO OTHER TARGETS. WITH THE MATURITY OF C COMPILER TECHNOLOGIES, AND NEWER CPU ARCHITECTURES THAT ARE MORE SUITABLE FOR HIGH LEVEL LANGUAGE COMPILATION, THE QUALITY OF THE C COMPILER GENERATED CODE FOR THESE NEWER 8-BIT MICROCONTROLLERS CAN BE COMPETITIVE WITH PROGRAMS WRITTEN IN ASSEMBLY LANGUAGE. HOWEVER, SOMETIMES ONE NEEDS TO KNOW THE CHARACTERISTICS AND QUIRKS OF THE ARCHITECTURE AND THE COMPILER ONE IS USING TO ACHIEVE GOOD TO EXCELLENT CODE OPTIMIZATION. IN THIS PAPER, I WILL DESCRIBE SOME OF THE AREAS THAT YOU MAY WANT TO PAY ATTENTION TO REGARDING YOUR SELECTED ARCHITECTURE AND COMPILER.

IO Registers

A microcontroller has on-chip peripherals that dramatically decrease the amount of external components needed in a design. It may have general purpose IO, serial IO, ADC and sometimes even special purpose IO pins that support protocol such as the I²C bus, all built into the chip itself. Typically these peripherals present themselves as IO registers to the CPU – for example, to generate a high signal on an output pin, one usually only requires the CPU to write a "1" to the corresponding IO register bits.

Some CPU architectures have a separate IO space for these registers with special instructions to access them. Since there is no such concept as IO space in the C language per se, in these cases the C compiler provides an extension allowing access to these IO registers. For example, you may have seen something like

```
unsigned char porta @port 0x3B;
```

The "@port <address>" is an extension that presumably declares an IO port with name "porta" at location 0x3B. Another popular way to declare something similar is:

```
°°SFR porta = 0x3B;
```

In this example, "SFR" (probably standing for Special Function Register) is used to declare "porta" at the same address. Since extensions are defined by the compiler vendors, the syntax and semantics vary among different compilers, even for the same microcontroller target. As they are not part of Standard C, certain behaviors may be not well defined. For example, the compiler may or may not allow you to declare a pointer to these objects (sometimes this is disallowed because the target machine disallows indirect accesses to the IO registers).

Since they are not standard C objects, the compiler may also restrict which C operators you can use on them, and tone should consult the compiler manual to see whether a particular C operator is allowed.

Memory Pointers

Some CPU architectures put these IO registers in regular data space. Some, like the Atmel AVR, even allow you to address the IO registers using either special IO instructions or by treating them as data memory. In this case, something like the following works well:

```
#define PORTA       (*(volatile
unsigned char *)0x3B)
…
unsigned char i = PORTA;  //
read
…
PORTA = i;          // write
```

The #define expression casts location 0x3B as a pointer to an unsigned char and then dereferences it, allowing the expression to be used to both read and write to the address. The "volatile" qualifier tells the compiler that the object at this location may change, so that compiler should always perform an actual read or write to the location and not use any previously cached values. If a CPU architecture allows you to access the IO registers as memory addresses, then you can treat them exactly like any other memory pointers. This allows you to perform the full range of C operations on the objects.

Accessing Bits

One often needs to access individual bits of an IO register. For example, PORT A may be an 8-bit output register, and each bit of the IO register corresponds to a hardware output pin of the PORT. There are several methods of accessing bits in C, with advantages and disadvantages to each approach:

Bitwise Operations

Plain C is powerful enough to perform any needed bit operations on IO registers (or any other integer objects). (After all, one of the first major tasks for the original C compiler was to rewrite the nascent Unix operating system in C!) Note the following bitwise operation example:

```
#define PORTA       (*(volatile
unsigned char *)0x3B)

#define BIT(x)      (1 << (x))
// bit position
…
PORTA |= BIT(0);          //
turn on 0th bit

PORTA &= ~BIT(0);   // turn off
0th bit

PORTA ^= BIT(0);          //
toggle the 0th bit
```

```
if (PORTA & BIT(0))// test to
see if 0th bit is set
…
```

This approach is probably the best overall: it works on all compilers, it defines the bit position explicitly and without ambiguity, and it will often result in optimal code sequences from the compilers. A minor inconvenience is that the usage seems more awkward than using bitfield names (as described below), but this can be alleviated by using C preprocessor macros; for example:

```
#define SETBIT(p, b)      (p)
|= BIT(b)

#define CLRBIT(p, b)      (p)
&= ~BIT(b)
etc.
```

Bitfields in a C Struct

C allows you to declare bitfields within a structure, such as:

```
typedef struct {

      unsigned      bit0 : 1,
                    bit1 : 1,
                    bit2 : 1,
                    bit3 : 1,
                    bit4 : 1,
                    bit5 : 1,
                    bit6 : 1,
                    bit7 : 1}
IOREG;
#define PORTA(*(IOREG *)0x3B)
…
int i = PORTA.bit0;// read
…
PORTA.bit0 = i;           //
write
```

Again, we see that it is more convenient if the CPU allows IO registers to be treated as data memory. Casting the IO location (0x3B in this example) to the appropriate structure type is no different from casting it as a pointer to a byte. If you must use an extension such as "@port" or "SFR", as shown earlier, you may or may not be able to declare bitfield structures and use them as described.

This is seemingly a nice way to map the IO register bits to the C language. However, a potential problem exists: the C Standard does not define the bitfield allocation order, and the compiler may allocate bitfields either from right to left or from left to right. If you use this method, you should make sure to consult the compiler manual to ensure that your use of the bit ordering matches the compiler's usage. It is also possible that some compilers may generate more verbose code for bitfield operations as compared to bitwise operations. Lastly, according to the C Standard, only "unsigned (int)" and "int" are acceptable datatypes for a bitfield member. Some compilers allow "unsigned char", but it is an extension. Whether or not a compiler allocates only a byte for the above structure depends on the particular compiler. If a compiler uses two bytes for the above structure, then using this method of accessing bits will not work. Due to these reasons, this approach is not really recommended for bitwise accessing of IO registers.

A similar situation applies to multi-byte registers such as the Atmel AVR ADC register pair. It consists of a high and low data register that have consecutive addresses, but which need to be accessed in certain order. Make sure the compiler does this properly, or if you roll your own code, make sure YOU do it properly.

IO Port Bit Extension

Some compilers that provide the IO register syntax extension (e.g. "SFR" declaration) may further provide an extension to specify the bit position of a named IO register. For example:

```
SFR PORTA = 0x3B;
…
i = PORTA.0;        // read the
0th bit
PORTA.0 = 1; // set the bit
…
etc.
```

In other words, the operator ".<digit>", which is an extension to Standard C, allows you to access the bit denoted by the digit. Unlike structure bitfield, the bit position is explicit and unambiguous. However, since this is an extension and since a good solution already exists using standard C bitwise operations, this method is not recommended.

Const Qualifier and Strings in Harvard Architecture

If you have read-only tables or "variables", then you should declare them with the "const" modifier. In most cases, the compiler will allocate them in the program memory and not take away precious SRAM space. Some microcontrollers have what is known as the "Harvard Architecture" – the program and data spaces are separate and different instructions are needed to access items in the separate spaces. The normal semantic of C literal strings (e.g. "strings") is that they must behave like arrays in data space. Consider the case with the string function strcpy: You should be able to call the function with the second argument being either a literal string or an array in RAM. However, using this takes up valuable RAM space. To lessen the use of the precious SRAM, some compilers for Harvard Architecture targets allow you to make strings allocatable in the program space, However, selecting this option means that you will probably need to call different functions depending on whether the argument is a literal string or an array in the data space.

Global Variables or Local Variables?

In theory, the choice of whether to declare a variable as a global or local variable should be dictated by how it is used. If the variable is accessed by multiple functions spread across different files, then they should be declared as global variables, e.g. declared outside of any function definitions. If a variable is used only within a function, then it should be declared inside the function as a local variable.

To further limit the visibility of the variable name and thus improve program readability, if a global variable is accessed only by the functions within a single file, you can prefix the variable declaration with the storage class "static" to make it visible to that file only. When a variable is only used with a statement block (inside a function) but its value must be retained across multiple invocations of the function, then you should declare the variable with the "static" storage class in the statement block where it is used. (This helps to further limit name visibility.) Despite the differences in syntax, file-static and function-static variable still behave like global variables and are treated as such. Some 8-bit systems have separate memory spaces, e.g. 8051 has internal and external data space. This may limit how you declare and use global variables.

If a variable is only used within a statement block and does not need to retain its value across multiple function invocations, then it is declared with the "auto" storage class in the statement block where it is used. The auto storage class is the default storage class for any variable declared inside a statement block, so you may omit it, or you may explicitly use the keyword "auto" to specify the storage class. The keyword "register" has the same meaning as "auto", except that you are providing a hint to the compiler that it should try to allocate this variable to a CPU register (although the compiler is free to ignore the hint), and you will not be taking the address of a register variable. The compiler allocates storage for global variables at program link time, and therefore each global variable has a unique address in the SRAM. The instructions that access global variables typically encode the addresses as part of the instructions. Since an address is usually 16 bits long in an 8-bit

microcontroller, each global variable address takes up to 2 bytes in the instruction. Some CPU architectures allow a short form of addressing, using only one byte to encode a global variable address if it fits certain conditions.

The usage patterns of local variables mirror a stack: as the function becomes active, the function's local variables become active as well. Once the function exits, the function's local variables can be destroyed. If the target architecture provides support for a stack, then the compiler will probably use the stack for allocation of the local variables. A nice feature of the stack is that the maximum amount of memory used for local variable allocation is usually less than the total number of local variables in the program, since stack space is reclaimed once a function exits. Unfortunately, some popular 8-bit microcontrollers do not support a stack or support only a limited version of a stack. In those cases, the compiler typically examines the usage patterns of the local variables and allocates them statically (possibly assigning some of them to the same addresses), simulating the natural effect performed by the stack.

If your chosen microcontroller does not directly support a stack, then you should merely declare your variables in the usual way and not worry about optimizing their usage. However, if your chosen CPU architecture does support a stack, then you may wish to examine how your compiler generates code for global and local variables, and see whether there are benefits in favoring one type of variable over the other, because of the CPU instruction set and memory. Note that this sort of optimization should not be undertaken casually, since for program readability variables should be declared in a manner consistent with their use.

As a test, you can use this simple program:

```
void main(void)
      {
…
      a = b + c;
      }
```

Declare the three operands as global variables and then as local variables, and note the differences in the sizes of the resulting code. If the target architecture supports short and long forms of global variables, declare them as such and see the differences produced there too. If you are declaring them as local variables, you may have to initialize them, possibly using a function call.

Otherwise, an optimizing compiler may eliminate some or all operations:

```
int foo(int); main()
      {
      int a, b = foo(2), c =
foo(3);
      a = b + c;
      foo(a);
// uses "a" so the compiler
will not optimize it away
      }

int foo(int x) { return x; }
```

With an aggressive compiler, the above program may still be optimized away, but chances are that it will produce usable results for most compilers.

It is best to look at the compiler-generated listing and not just at the total memory usage from a map file, as there will be extra code that the compiler puts in to make your program into a complete executable program. Most CPU architectures will use more instruction space to access global variables. How much more depends on the architecture and the compiler. If the code bloat associated with using global variables is acceptable to you, then by all means do not attempt to optimize their usage.

Sometimes it is even possible to save code space by using global variables; for example, as an alternative to passing parameters between functions (which can be expensive under some CPU architectures). This is a fairly controversial subject. If you go with this route, be sure to observe good software engineering practices, e.g. give the variables good descriptive names, limit and localize their accesses, etc.

So, what can you do to reduce the overhead of using global variables? You might rewrite your code so that global variables are not used. You should look at the resulting program to make sure that you do not add code bloat elsewhere, since you may have to change your program algorithmically. Another possibility is that you can cache global variable accesses:

```
{
extern int global_i;
int i = global_i;
…
// read/write using "i" instead
of "global_i"
…
global_i = i;
}
```

This only works if any updated values of "global_i" are not needed in an interrupt handler or something similar while this function is executing. Before making this change, you should check how your compiler handles multiple appearances of the same global variable in a function. Some compilers may perform similar caching of the global variables as in the example above, saving you from doing it manually. Some compilers may also cache a pointer to the global variable, which may still be a win under some architectures, and this technique allows asynchronous concurrent access of the global variable in an interrupt handler.

Register Promotion Optimizations

Most of the earlier microcontrollers have few registers; some as little as a single accumulator. Some of the newer microcontrollers have more registers. For example, the Atmel AVR has 32 8-bit registers, and all of the AVR arithmetic and logical instructions will only work on register operands. Even if a microcontroller can operate on memory operands, it often still pays to keep the operands in registers since they take up less instruction space.

The C storage class "register" is meant to be a hint to the compiler that the variable should be allocated to a CPU register if possible, instead of in the default stack location. Fortunately, most modern compilers now take care of this automatically by performing register allocation optimization, and the "register" keyword is usually not necessary. Again, consult your compiler manual to see whether this is done automatically or not.

The quality of the register allocation varies as well, depending on the compiler and the specific architecture. The more general the register set, the easier it is for good register optimizations. Here are some register related optimizations that a compiler may perform, beyond the automatic promotion of certain variables into registers:

1. Lifetime Analysis – an optimization that determines the beginning and end of the variable usage. This analysis must handle loops and other control structures in the programs. Using this information, further optimizations can be performed.

2. Lifetime Splitting – using the lifetime analysis information, the compiler may "split" a variable into multiple pseudo-variables, each with distinct lifetimes.

3. Register Allocation – using lifetime information, the compiler can pack the variables into registers more intelligently. For example, it may pack multiple variables into a single register if their lifetimes do not overlap. Or if it has performed lifetime splitting, then each distinct lifetime may get its own register.

The more registers available to the compiler, the better it can do register allocation. Compilers that do lifetime splitting, for example, can allow the compiler to allocate the split pseudo-variables into different registers, or perhaps put only some of them in registers, depending on the situation. If your compiler does not perform lifetime splitting, then declaring the variables in the smallest enclosing statement block and using distinct variables for different uses will help the compiler to do a better job.

Some compilers allow you to declare global registers, thus allowing you to assign a certain number of global variables to registers. This is especially useful if your program uses a lot of interrupt handlers. Normally, if you write an interrupt handler in C (via some sort of extension since Standard C has no syntax for declaring interrupt handlers), then the compiler must save and restore any registers the handler uses, so that the state of the machine is restored when the handler exits. If there are a lot of interrupts or interrupt handlers, it can be costly in both instruction size and the speed of the program. Using global registers solves this problem.

Choose Your Data Types Carefully

An advantage of using C over assembly language is the data type handling – you simply declare variables of the needed types, and the compiler takes care of storage allocation and code generation. However, if you are looking to optimize your program, you need to be careful what data types you use for your variables. Standard C does not dictate the sizes of the integer data types except that the following size relationship holds, and that "int" is at least 16-bits:

```
sizeof (char) <= sizeof (short)
<= sizeof (int) <= sizeof (long)
```

In practice, for most 8-bit C compilers, "char" is 8-bits, "short" and "int" are 16-bits, and "long" is 32-bits.

Using the "unsigned" type may possibly improve your code size, as some CPU architectures favor the use of unsigned types. For example, it may be cheaper to zero-extend a byte than sign-extend a byte into an integer word. Moreover, dividing an unsigned integer by a power of 2 can be done as a logical right shift, but a signed integer divide cannot be done by arithmetic right shifts without additional compensating code.

You should avoid using "long" or floating point variables unless necessary, because most - if not all - 8 (and even 16-bit) microcontrollers do not support these operations directly. In fact, most 8-bit microcontrollers do not support 16-bit operations at all. Using long data types in an expression will probably increase the code size and the running time of the code by at least a factor of two to four compared to using "int" data types, depending on which C operations you use.

Standard C provides two floating point data types: float and double (long double was introduced to the standard in C99). Most C compilers provide at least the 32-bit "float" data type, usually conforming to various degrees to the IEEE754 standard. The C Standard dictates that the "double" data type be at least 64-bits, but some compilers opt to make "double" 32-bits, since the need for 64-bit floating point is very rare on an 8-bit microcontroller.

Using floating point operations will dramatically increase the code size and execution speed of your program. In fact, some people argue that floating point should not be used unless necessary in the 8-bit microcontroller world because of the size and speed penalty and the intrinsic imprecise nature of the floating point operations. For example, in this simple program:

```
float f = 3.1;

f -= 1.0;

if (f == 2.1) do_something();
```

The comparison may not execute as true since floating point computation depends on the implementation. Any implementation must compromise on either the range or the precision of the floating point computations. For the purpose of optimization and perhaps even for the purpose of a better fit to your task, you may wish to investigate using alternatives to floating point. Sometimes one can use plain integers, and sometimes one has to use scaled integers. Unfortunately, since scaled integers and other similar alternatives are not in Standard C, any such use would either be provided by the compiler as an extension, as library functions, or as a roll-your-own solution.

Overhead of Library Functions

One word: printf. While it is expected that the compiler provide a printf function, a full implementation supporting all the features can eat up a lot of program memory. There are a lot of esoteric features in printf, plus a full implementation drags in all the floating point support functions. Most compilers provide you the option of selecting different versions of printf, each with varying capabilities and code size requirement. Choose the one that closest matches your needs. If your program space or SRAM space is really tight, you may perhaps even forgo printf and use the simpler conversion functions such as itoa(), ltoa(), and ftoa() if your compiler provides them.

Conclusion

Since the cost of an embedded system is magnified by the number of units shipped, embedded engineers need to juggle between the cheapest possible chips, development cost, and the time-to-market. Well-written C helps to give you a leg up on the competition when working on the next version of your product, as C allows more control over "almost-always-not-enough" scarce resources. Hopefully, attention to and understanding of the issues brought up in this paper will assist in your 8-bit microcontroller C programming endeavors. ■