AN2093

Creating Efficient C Code for the MC68HC08

by Stuart Robb
East Kilbride, Scotland

1 Introduction

The C programming language is a powerful, flexible and potentially portable high-level programming language. These and other features, such as support for low-level operations, make this a useful language for programming embedded applications. Many embedded applications use low-cost microcontrollers with an 8-bit data bus. Such microcontrollers often have limited on-chip resources, such as few CPU registers and limited amounts of RAM and ROM. Compared to other 8-bit microcontrollers, the HC08 architecture is well suited to the C programming language. It has an effective instruction set with addressing modes which enable efficient implementation of C instructions. The instruction set includes instructions for manipulating the stack pointer. The addressing modes include indexed addressing modes with the index contained in the index register or the stack pointer register. These features allow efficient access to local variables. The C language may be used successfully to create the program for the HC08 microcontroller, but to produce the most efficient machine code, the programmer must carefully construct the C language program. In this context, “efficient code” means compact code size and fast execution time. The programmer must not only create an efficient high level design, but also pay attention to the detailed implementation. Principally, efficiency improvements may be obtained by appropriate design of data structures and use of data types. Programmers accustomed to coding in assembly language will be familiar with these issues and C programmers should remember that their C code is converted into assembly language by the compiler. In actual fact, the compiler will recognise certain constructs in C and replace them with functions or in-line code which have often been hand coded in assembly. Thus a

compiler is no more efficient that a good assembly programmer. It is however, much easier to write good code in C which can be converted into efficient assembly code than it is to write efficient assembly code by hand. The potential drawback is that it is very easy to create C code which no compiler, no matter how good, can convert into efficient assembly code.

Some hints and tips are presented in this paper to aid the programmer to write their code in C in a way in which a compiler can convert into efficient machine code. Some of these tips will also improve the portability of the code. Examples are given based on real compiler output generated by the Hiware HC08 compiler from Metrowerks Europe (formerly Hiware AG), a Freescale company.

2 CPU08 Register Model

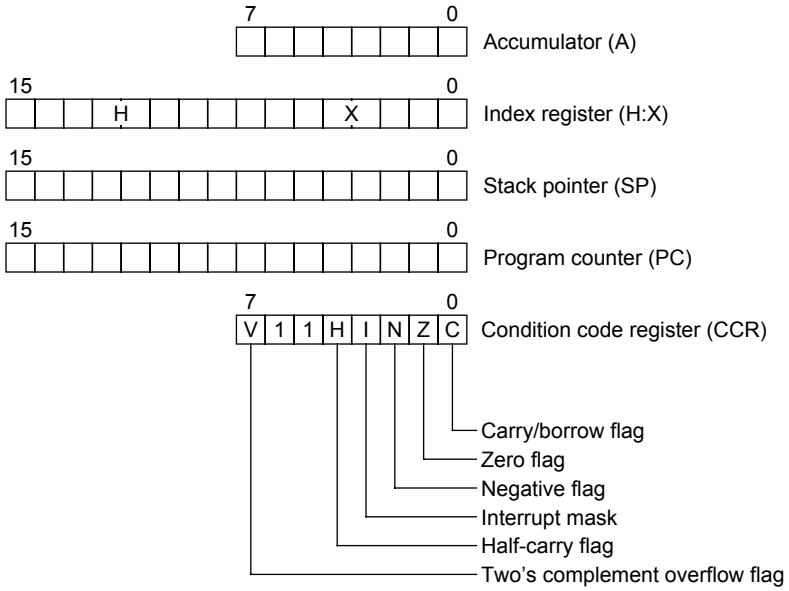

Figure 1 CPU08 Register Model

An overview of the CPU08 register model is given here for completeness. The CPU08 has five registers which are not part of the memory map. These registers are briefly described.

Accumulator The accumulator is a general purpose 8-bit register. The CPU uses the accumulator to hold the operands and results of operations.

Index Register The 16-bit index register is called H:X and is used by indexed addressing modes to determine the effective address of an operand. The index

register can access a 64K byte address space in this mode. The lower byte X is used to hold the operand for the MUL and DIV instructions. H:X can also serve as a temporary data storage location.

Stack Pointer The 16-bit stack pointer is used to hold the address of the next available location on the stack. The CPU uses the contents of the stack pointer register as an index to access operands on the stack in stack pointer offset addressing modes. The stack can be located anywhere where there is RAM in the 64K byte address space.

Program Counter The 16-bit program counter contains the address of the next instruction or operand to be fetched. The program counter can access a 64K byte address space.

Condition Code Register The 8-bit condition code register contains the global interrupt mask bit and five flags that indicate the results of the instruction just executed. Bits 5 and 6 are permanently set to logic 1.

3 Addressing Modes

The CPU08 has 16 different addressing modes. A brief overview is given here.

Inherent Inherent instructions have no operand to fetch and require no operand address. Most are one byte long.

Immediate The operand for immediate instructions is contained in the bytes immediately following the opcode. Immediate instructions therefore have constant operands.

Direct Direct instructions are used to access operands in the direct page, i.e. in the address range $0000 to $00FF. The high-order byte of the address is not included in the instruction, thus saving one byte and one execution cycle compared to extended addressing.

Extended Extended instructions can access operands at any address in a 64K byte memory map. All extended instructions are three bytes long.

Indexed Indexed instructions use the contents of the 16-bit index register to access operands with variable addresses, such as variables accessed

through a pointer. There are five modes of indexed addressing: indexed, no offset, with or without a post-increment of the index register; indexed, 8-bit offset, with or without a post-increment of the index register; and indexed with a 16-bit offset.

Stack Pointer Stack pointer instructions are similar to indexed instructions only they use the contents of the stack pointer as an address of the operand instead of the index register. There are two modes of stack pointer addressing: stack pointer with an 8-bit offset and with a 16-bit offset. Stack pointer instructions require one extra byte and one extra execution cycle compared to the equivalent indexed instruction.

Relative All conditional branch instructions use relative addressing evaluate the effective address. If the branch condition is true, the CPU evaluates the branch destination by adding the signed byte following the opcode to the program counter. The branch range is –128 to +127 bytes from the address after the branch instruction.

Memory to Memory Memory to memory instructions copy data from one location to another. One of the locations is always in the direct page. There are four modes of memory to memory instructions: move immediate data to direct location; move direct location to direct location; move indexed location to direct location with post-increment of the index register; and move direct location to indexed location with post-increment of the index register.

4 Basic Data Types

Table 1 Basic Data Types

Data Type	Size	Range (unsigned)	Range (signed)
char	8 bits	0 to 255	–128 to 127
short int	16 bits	0 to 65535	–32768 to 32767
int	16 bits	0 to 65535	–32768 to 32767
long int	32 bits	0 to 4294967295	–2147483648 to 2147483647

Easily the greatest savings in code size and execution time can be made by choosing the most appropriate data type for variables. This is particularly true for 8-bit microcontrollers where the natural internal data size is 8-bits (one byte) whereas the C preferred data type is ‘int’. The ANSI standard does not precisely define the size of its native types, but

compilers for 8-bit microcontroller's usually implement 'int' as a signed 16-bit value. As 8-bit microcontrollers can process 8-bit data types more efficiently than 16-bit types, 'int' and larger data types should only be used where required by the size of data to be represented. Double precision and floating point operations are particularly inefficient and should be avoided wherever efficiency is important. This may seem obvious, but it is often overlooked and has a huge impact on code size and execution time.

As well as the magnitude of the required data type, the signedness must also be specified. The ANSI standard for C specifies 'int' to be signed by default, but the of 'char' is not defined and may vary between compilers. Thus to create portable code, the data type 'char' should not be used at all. Instead the signedness should be defined explicitly: 'unsigned char' or 'signed char'. It is good practice to create type definitions for these data types in a header file which is then included in every other file. It is also worthwhile to create type definitions for all the other data types which are used as well, for consistency, and to allow for portability between compilers. Something like the following may be used:

```
typedef unsigned char UINT8;
typedef   signed char SINT8;
typedef unsigned int  UINT16;
typedef          int  SINT16;
typedef unsigned long int UINT32;
typedef          long int SINT32;
```

A variable is typically used in more than one expression, but some of those expressions may not require the full data size or signedness of the variable. In this case, savings can be made by casting the variable, or part of the expression containing the variable, to the most appropriate data size.

Summary:

- Create type definitions for all data types used.
- Use the smallest data type appropriate to each variable.
- Use signed data types only when required.
- Use casts within expressions to reduce data types to the minimum required.

5 Local versus Global Variables

Variables can be classified by their scope. Global variables are accessible by any part of the program and are allocated permanent storage in RAM. Local variables are accessible only by the function within which they are declared and are allocated storage on the stack. Local variables therefore only occupy RAM while the function to which they belong is running. Their absolute address cannot be determined when the code is compiled and linked so they are allocated memory relative to the stack pointer. To access local variables the compiler may use the stack pointer addressing mode. This addressing mode requires one extra byte and one extra cycle to access a variable compared to the same instruction in indexed addressing mode. If the code requires several consecutive accesses to local variables, the compiler will usually transfer the stack pointer to the 16-bit index register and use indexed addressing instead. The 'static' access modifier may be used with local variables. This causes the local variable to be permanently allocated storage in memory, like a global variable, so the variable's value is preserved between function calls. However the static local is still only accessible by the function within which it is declared.

Global variables are allocated permanent storage in memory at an absolute address determined when the code is linked. The memory occupied by a global variable cannot be reused by any other variable. Global variables are not protected in any way, so any part of the program can access a global variable at any time. This gives rise to the issue of data consistency for global variables of more than a single byte in size. This means that the variable data could be corrupted if part of the variable is derived from one value and the rest of the variable is derived from another value. Inconsistent data arises when a global variable is accessed (read or written) by one part of the program and before every byte of the variable has been accessed the program is interrupted. This may be due to a hardware interrupt for example, or an operating system, if one is used. If the global variable is then accessed by the interrupting routine then inconsistent data may result. This must be avoided if reliable program execution is desired and this is often achieved by disabling interrupts while accessing global variables.

The 'static' access modifier may also be used with global variables. This gives some degree of protection to the variable as it restricts access to the variable to those functions in the file in which the variable is declared.

The compiler will generally use the extended addressing mode to access global variables or indexed addressing mode if they are accessed though a pointer. The use of global variables does not generally result in significantly more efficient code than local variables. There are some

limited exceptions to this generalisation, one being when the global variable is located in the direct page.

However, use of global variables prevents a function from being recursive or reentrant, and often does not make the most efficient use of RAM, which is a limited resource on most microcontrollers. The programmer must therefore make a careful choice in deciding which variables, if any, to make global in scope. Worthwhile gains in efficiency can sometimes be obtained by making just a few of the most intensively used variables global in scope, particularly if these variables are located in the direct page.

Summary:

- Careful analysis is required when deciding which variables to make global in scope.

6 Direct Page Variables

The address range $0000 to $00FF is called the direct page, base page or zero page. On the M68HC08 microcontrollers, the lower part of the direct page always contains I/O and control registers and the upper part of the direct page always contains RAM. After a reset, the stack pointer always contains the address $00FF. The direct page is important because most CPU08 instructions have a direct addressing mode whereby they can access operands in the direct page in one clock cycle less than in extended addressing mode. Furthermore the direct addressing mode instruction requires one less byte of code. A few highly efficient instructions will only work with direct page operands. These are: BSET, BCLR, BRSET and BRCLR. The MOV instruction requires one of the operands to be in the direct page.

A compiler cannot take advantage of the efficient direct addressing mode unless variables are explicitly declared to be in the direct page. There is no ANSI standard way of doing this and compilers generally offer different solutions. The Hiware compiler uses a #pragma statement:

```
#pragma   DATA_SEG SHORT myDirectPageVars
UINT16    myDirectPageVar1;     /* unsigned int in direct page */
#pragma   DATA_SEG DEFAULT
```

This declares the direct page segment myDirectPageVars which contains the variable myDirectPageVar1 which may be accessed using the direct addressing mode. The programmer must remember to make the linker place the myDirectPageVars segment at an address in the

direct page. The amount of RAM in the direct page is always limited, so only the most intensively used variables should be located in the direct page. To release more of the direct page RAM for global variables, the stack can be relocated to RAM outwith the direct page, if available. This will not affect the stack pointer addressing modes.

Many I/O and control registers are located in the direct page and they should be declared as such so the compiler can use the direct addressing mode where possible. Two possible ways to do this are:

Define the register name and its address together:

```
#define PortA  (*((volatile UINT8 *)(0x0000)))
#define PortB  (*((volatile UINT8 *)(0x0001)))
```

or define the register names in a direct page segment and define the segment address at link time:

```
#pragma DATA_SEG SHORT myDirectPagePortRegisters
volatile UINT8 PortA;
volatile UINT8 PortB;
#pragma DATA_SEG DEFAULT
```

Summary:

- Declare all direct page registers to the compiler.
- Put only the most frequently used variables in the direct page.
- Release more direct page RAM for variables by relocating the stack.

7 Loops

If a loop is to be executed less than 255 times, use 'unsigned char' for the loop counter type. If the loop is to be executed more than 255 times, use 'unsigned int' for the loop counter. This is because 8-bit arithmetic is more efficient than 16-bit and unsigned arithmetic is more efficient than signed.

If the value of the loop counter is immaterial, it is more efficient to decrement the counter and compare with zero than to increment and compare with a non-zero value. This optimisation is not effective if the loop must be executed with the loop counter equal to zero, such as when the loop counter is used to index an array element and the first element must be accessed.

If the loop counter is used in expressions within the loop, remember to cast it to the most appropriate data type each time it is used.

When a loop is performed a fixed number of times and that number is small, such as three or four, it is often more efficient not to have a loop at all. Instead, write the code explicitly as many times as required. This will result in more lines of C code but will often generate less assembly code and may execute much faster than a loop. The actual savings will vary, depending on the code to be executed.

In summary

- Use the smallest appropriate unsigned type for the loop counter.
- Decrement the loop counter and compare with zero where possible.
- If the loop counter is used within the loop, cast it to the most appropriate data type.
- Do not use a loop if code is to be executed a small, fixed number of times.

8 Data Structures

When programming in C it is easy to create complex data structures, for example an array of structures with each structure containing a number of different data types. This will produce complex and slow code on a 8-bit microcontroller which has a limited number of CPU registers to use for indexing. Each level of de-referencing will result in a multiplication of the element number by the element size, with the result probably pushed onto the stack in order to do the next calculation. Structures should be avoided where possible and the data structures kept simple. This can be done by organising data into simple one-dimensional arrays of a simple data type. This will result in a greater number of arrays, but the program will be able to access the data much more quickly. If structures are unavoidable, they should not be passed as a function argument or a function return value, they should be passed by reference instead.

Summary

- Do not use complex data structures.

9 Examples

Examples of assembly code generated by the Hiware HC08 compiler are described in this section, based on the following type definitions:

```
typedef unsigned char UINT8;
typedef   signed char SINT8;
typedef unsigned int  UINT16;
typedef          int  SINT16;
```

9.1 Register1

This example illustrates bit manipulation for a register in the direct page (PORTA) and for one not in the direct page (CMCR0). Setting or clearing one or more bits in a register not in the direct page requires 7 bytes of ROM and 9 CPU cycles. Setting or clearing multiple bits in a register in the direct page requires 6 bytes of ROM and 8 CPU cycles. Setting or clearing a single bit of a register in the direct page requires 2 bytes of ROM and 4 CPU cycles.

C Code	Assembly Code	Bytes	Cycles

`#define PORTA (*((volatile UINT8 *)(0x0000)))`
`#define CMCR0 (*((volatile UINT8 *)(0x0500)))`
 	
`void`
`register1(void)`			

`{`	`LDHX #0x0500`	3	3
	`LDA ,X`	1	2
`CMCR0 &= ~0x01; /* clr bit1 */`	`AND #0xFE`	2	2
	`STA ,X`	1	2
`PORTA |= 0x03; /* set b1,2 */`	`LDA 0x00`	2	3
	`ORA #0x03`	2	2
	`STA 0x00`	2	3
`PORTA &= ~0x02; /* clr bit2 */`	`BSET 0,0x00`	2	4
`}`	`RTS`	1	4

9.2 Datacopy1

This is an example of the inappropriate use of ‘int’ for variable ‘i’ which is used as both the loop counter and the array index. The compiler is forced to use signed 16-bit arithmetic to calculate the address of each element of dataPtr[]. This routine requires 50 bytes of ROM and with 4 iterations of the loop, executes in 283 CPU cycles.

C Code	Assembly Code	Bytes	Cycles
`UINT8 buffer[4];`			
`void`	PSHA	1	2
`datacopy1(UINT8 * dataPtr)`	PSHX	1	2
`{`	AIS #-2	2	2
`  int i;`	TSX	1	2
	CLR 1,X	2	3
`  for (i = 0; i < 4; i++)`	CLR ,X	1	2
`  {`	TSX	1	2
`  buffer[i] = dataPtr[i];`	LDA 3,X	2	3
`  }`	ADD 1,X	2	3
`}`	PSHA	1	2
	LDA ,X	1	2
	ADC 2,X	2	3
	PSHA	1	2
	PULH	1	2
	PULX	1	2
	LDA ,X	1	2
	TSX	1	2
	LDX ,X	1	2
	PSHX	1	2
	LDX 3,SP	3	4
	PULH	1	2
	STA buffer,X	3	4
	TSX	1	2
	INC 1,X	2	4
	BNE *1	2	3
	INC ,X	1	3
	LDA ,X	1	2
	PSHA	1	2
	LDX 1,X	2	3
	PULH	1	2
	CPHX #0x0004	3	3
	BLT *-39	2	3
	AIS #4	2	2
	RTS	1	4

9.3 Datacopy2

In this example, the loop counter and array index variable is optimised to an ‘unsigned char’. This routine now requires 33 bytes of ROM, 17 bytes less than Datacopy1. With four iterations, Datacopy2 executes in 180 CPU cycles, 103 cycles less than Datacopy1. In this example the value of the loop counter is important; the loop must execute with i = 0, 1, 2 and 3. No significant improvement is obtained in this case by decrementing the loop counter instead of incrementing. Also, no significant improvement is obtained in this case if variable ‘buffer’ is place in the direct page: the instruction ‘STA buffer,X’ uses direct addressing instead of extended, saving one byte of code and one CPU cycle per iteration.

C Code	Assembly Code	Bytes	Cycles
`UINT8 buffer[4];`			
`void`			
`datacopy2(UINT8 * dataPtr)`	PSHA	1	2
`{`	PSHX	1	2
`    UINT8 i;`	PSHH	1	2
	TSX	1	2
`    for (i = 0; i < 4; i++)`	CLR ,X	1	2
`    {`	LDA ,X	1	2
`        buffer[i] = dataPtr[i];`	ADD 2,X	2	3
`    }`	PSHA	1	2
`}`	CLRA	1	1
	ADC 1,X	2	3
	PSHA	1	2
	PULH	1	2
	PULX	1	2
	LDX ,X	1	2
	TXA	1	1
	TSX	1	2
	LDX ,X	1	2
	CLRH	1	1
	STA buffer,X	3	4
	TSX	1	2
	INC ,X	1	3
	LDA ,X	1	2
	CMP #0x04	2	2
	BCS *-25	2	3
	AIS #3	2	2
	RTS	1	4

9.4 Datacopy3

In this example, the data is copied without using a loop. This routine requires 23 bytes of ROM and executes in 36 CPU cycles. This is 10 bytes less ROM and 144 fewer CPU cycles than Datacopy2. If 8 bytes were to be copied, this method would require 10 bytes more ROM than Datacopy2, but would execute in 280 fewer CPU cycles. Although there are potential savings to be had if the variable ‘buffer’ is located in the direct page, the compiler does not take full advantage of them in this case.

C Code	Assembly Code	Bytes	Cycles
`UINT8 buffer[4];`			
`void`			
`datacopy3(UINT8 * dataPtr)`	PSHX	1	2
`{`	PULH	1	2
`buffer[0] = dataPtr[0];`	TAX	1	1
`buffer[1] = dataPtr[1];`	LDA ,X	1	2
`buffer[2] = dataPtr[2];`	STA buffer	3	4
`buffer[3] = dataPtr[3];`	LDA 1,X	2	3
`}`	STA buffer:0x1	3	4
	LDA 2,X	2	3
	STA buffer:0x2	3	4
	LDA 3,X	2	3
	STA buffer:0x3	3	4
	RTS	1	4

9.5 Loop1

If only the number of iterations matter and not the value of the loop counter, it is more efficient to decrement the loop counter and compare it with zero. In this example the 'for' statement requires 7 bytes of ROM. and the increment and test of the loop counter take 6 CPU cycles for each iteration. This saves 2 bytes of ROM and 9 CPU cycles per iteration compared to the 'for' statement in Datacopy2. However this optimisation cannot be applied to Datacopy2 as the code within this loop is executed with i = 4, 3, 2 and 1.

C Code	Assembly Code	Bytes	Cycles
void loop1(void) {			
UINT8 i;	PSHH	1	2
	LDA #0x04	2	2
for(i=4; i!=0; i--)	TSX	1	2
{	STA ,X	1	2
/* code */			
}	TSX	1	2
}	DBNZ ,X,*-offset	2	4
	PULH	1	2
	RTS	1	4

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor