CHAPTER 5

Compiler Runtime Architecture & C Programming Topics

Introduction

This chapter describes the software conventions that the ICC11 compiler uses.

Implementation Characteristics

Base Data Type

Char data are l byte each, short and int are 2 bytes. Long data type are 4 bytes each. Char data type is equivalent to the signed char data type. Float and double are 4 bytes each. using the IEEE single precision format with I bit of sign, 8 bits of exponent biased with a value of 127, and 23 bits of mantissa.

You must declare a function that accepts a non-int argument (including float, double, long, and struct) or returns a non-int result before calling it since these function have different calling conventions. For example, you must include the file math. h before calling any of the floating point library functions and s tdl ib . h if you use the atof() function.

The floating point support library uses global data and thus is not reentrant. If you use floating point code under a multitasking environment, you must ensure that a floating point computation finishes if it may be preempted by another floating point computation.

Global and Static Data

"Uninitialized" C global data are put in the bss section, and explicitly initialized global and static data are put in the data section. The start-up routine initializes the bss section to zero when the program is run. Since the data section is typically in RAM, for each entry in the data section, there is a corresponding entry in the idata section with the initialized value. The start-up routine copies the bytes from the idata area to data area so the global data would have the correct values even after a power reset. The idata area must **be placed in** non-volatile **memory and** is loaded **after the text area by default.**

Note: only items in the data area with the default name "data" gets initialized by the start-up code. So if you change the data area name. those data will not get initialized by entries in the idata area!

If you have initialized constant data that you don't modify, you may put them in the text section in which case they would not occupy space in the data and idata areas. You may do this by specifying the data as const in C:

```
const int table[] = {1. 2. 3) ; /* in .text */
int table2[] = (4, 5. 6); /* in .data */
```

Idata

As mentioned, idata contains the initial values of the global and static data in the area "data." Normally, you do not need to specify a start address for the idata area since it defaults to follow the text area. Since the startup code does byte-by-byte copying from the idata to data area, the idata area and the data area must each be contiguous. Thus if you use discontiguous area specifier for the text area (e.g. -btext:0x1400.0x3FFF:0x4001), then you will have to specify the start address for idata separately. Otherwise, the linker gives an error message.

Miscellaneous

External names are significant up to 32 characters.

Arguments are evaluated from right to left, and bitfields are allocated from left to right.

The compiler allocates an argument block for a function on function entry. Therefore it does not generate explicit argument pushings and poppings when a function is called. This generates shorter and faster code.

Local variables and function parameters are addressed via the X index register. Even though the HC11 has a limitation of only allowing 8 bit offset (256 bytes) from an index register, the compiler generates correct code to access items that are more than an 8 bit displacement away. However, it requires several instructions to do the extra computation, so if you are concerned about code size or speed, you may try not to use more than 256 bytes of local storage.

The X register is used as the frame pointer and must be restored to its original value at function exits.

Since floating point computation uses a fair amount of code space, it is likely that programs using floating point would not fit in a single chip mode system because such a system only contains a small amount of code storage.

Assembly Code and Calling Conventions

Interface With Assembly Routines

Time or space critical code can be written in assembler language routines or embedded assembly code in your C source (chapter 6 describes the assembler syntax and operations).

The compiler prepends an underscore to all global functions and data names. Thus, to access such objects in assembly code. you must prcpcnd an underscore to the name of the object:

```
        Idd      _foo ; load the C variable "foo"
```

Arguments are promoted to their natural sizes, and pushed from right to left on the stack, **except** for the first argument which is passed in the D register[1] unless the first argument is a structure, floating point, or long. The natural size for integer types, including char and short, is int. The natural size for floating point types is

1. To call a function that returns a structure or float. the compiler generates code to pass the address of a (temporary) structure using the D register. This structure holds the return value of the function when the function returns.

double, and the natural size for a structure is simply the size of the structure. On entry to a function not returning structure or floating point, the stack. looks like this:

```
argument 3
-----------------
argument 2
-----------------
return address
```

The stack pointer SP points to one bytc below the return address . A typical function prologue sequence pushes the first argument from the D registcr on thc stack, then pushes the previous frame pointer X on the stack, and finally sets thc X register to the current stack pointcr. It then adjusts the stack pointcr by the amount of local storagc the function needs and transfers thc stack pointer to index register X. All local variables and arguments arc refer-enced by using a displacement off the X rcgistcr.

A routinc docs not havc to prcscrvc any rcgistcrs except SP and X which must havc thc samc valucs when the function is cntcrcd. The D register is uscd to pass thc first argumcnt and to rcturn intcgcr valucs from thc function.

The two long "registcrs" arc allocatcd at -4 and -8 bytes below the frame pointer X. The two floating point "registers" are global variables and hence routincs that usc floating point arc non-reentrant.

For example, the following putchar( ) assembly routine takes a C character and calls a BUFFALO routine to output thc charactcr. Since BUFFALO expects to be called with the arguments in registers, this routine moves the argument to the right place before making thc actual call:

```
; void putchar(int c)
; output 'c' as an ASCII character using BUFFALO
        .text ; code
_putchar:   ; note the prepended underscore
        tba ; 'c' is at B, move to A
        jsr Oxffaf ; BUFFALO output routine
        rts
```

Embedded ASM Statements

In addition to linking with assembly modules, you may embed arbitrary assembly statements in your C programs. The format is:

```
asm("asm string");
```

The compiler inserts a tab at the beginning of each line and a newline at the end of the specified string, C escape sequences such as \n can also be used to embed a series of assembly instructions with a single asm call. Inside the supplied string, a reference in the form %<variable name>, where <variable name> is an in-scoped data variable, will be replaced by the assembly reference to the variable. Note that the peephole optimizer ignores the instructions inside an asm() statement. This means branches across asm() statements will always be long branches.

To generate multiple line embedded assembly statements, you may write multiple asm() calls, or use the '\n' escape character, combined with the string concatenation feature of ANSI C:

```
int i;
asm("ldd %i\n" /* note no comma */
  "std Ox 1000\n" /* %i will be replaced by */
  "bra .-4");   /* ?,x */
```

You can only use asm() in an expression statement context (i.e., only by itself and not as part of a larger expression), or outside a function definition. You must be careful to preserve the X register in embedded asm since it is the frame pointer.

HC11 Specific Functions

Accessing On-Chip Peripherals

On chip ROM, RAM and IO registers are addressed like all memory addresses.

One efficient way is to use pointer indirections. For example. to access PORTA at address Ox 1000, you write:

```
I* write one to port a */
*(volatile unsigned char *)0x 1000 = I.; /* or */
PORTA = 1; /* if chc 11.h> is included */

/* read content of port a to a variable */
```

```
i = *(volatile unsigned char *)0x1000; /* or */
i = PORTA;
```

The header file hc11.h includes macro definitions for the internal registers so they may be referred to using mnemonics such as PORTA. If you relocate the IO register block to other than the default base address, you will need to modify one entry in this file.

EEPROM is read like other memory locations. However, writing 10 EEPROM requires special timing routines. The function write_eeprom() gives you an example on how to program an EEPROM cell.

Some languages or pseudo-C systems require you to use routines such as PEEK or POKE to access memory. They are not necessary in C since C allows memory indirection as shown above.

Bit-Twiddling Examples

Often accessing and using the HC I l Integrated functions such as the timers and the AD converters involved "bit-twiddling" of the IO registers. In most cases, C allows you to write this kind of code with ease. Here are some examples:

```
DDRD = 0x0F; /* port D, bit 0 to 3 are output */
DDRD &= ~0x0F; /* Port D, turn off bit 0 to 3 as output */
TFLG2 = 0x80; /* CLEAR the TOF bit */
TFLG2 I= 0x80; /* enable the TOI */
```

Interrupt Routines

To implement an interrupt handler as a C function, you must do the following:

1. You must declare the function as an interrupt handler so that the compiler will generate a rti (return from interrupt) instead of the rts (subroutine return) instruction. You do this by using the following pragma:

   ```
   #pragma interrupt-handler <name> [<name>]*
   ```

   For example:

   ```
   #pragma interrupt-handler foo
   void foo() (/*this is an interrupt handler*/)
   ```

2. You must assign the function address to the interrupt vector. You either use vectors.c for a ROM based system, or assigning it at runtime for a RAM based system under monitor control. See “Interrupt Vectors” on page 28.

Example: Pulse Width Modulation Generator

As an example, here arc some code fragments to implement Pulse Width Modulation (PWM) using the Output Compare interrupt functions of the HC 11. PWM is useful for controlling devices such as servo motors. .

Each of the output compare functions has a 16 bit compare register and an output pin associated with it. In its simplest operation, whenever the value of the free running system clock matches the value in the compare register of a output compare function, its output pin goes to a preprogrammed state and an interrupt gets triggered.

Let’s define a system clock cycle as the time for the timer to start from 0 and overflows to 0 again. This is just 64K times the cycle time, or 32.768 ms. for a 2 Mhz system with a 8Mhz clock. In this example, we will generate a pulse width defined by the constant PULSE-WIDTH every system clock cycle. Wc will usc Output Compare function five, pin A3.

First, here is the interrupt function:

```
#pragma interrupt-handler PWMDriver
void PWMDriver()
        {
        static int state = 0;

        TFLG1 = 0x08;
        if (state == 0)
                {
                TOC5 = PULSE-WIDTH;
                TCTLI = (TCTLI & -0x3) | 0x1;
                }
        else
                {
                TOC5 = 0;
                TCTLI |= 0x3;
                }
            state ^= I;
            }
```

Basically, every time the interrupt handler function PWMDriver is called, it:

1. specifies when the next interrupt should happen,
2. what the output pin value should then be,
3. toggles a static variable so the two paths get executed alternately, and
4. returns with the rti instruction (done automatically by declaring this as an interrupt handler).

The first path sets up an interrupt to trigger at a time equal to PULSE-WIDTH. at which point the output pin goes low. The second path sets up an interrupt to trigger at time cqual to 0. at which point the output pin goes high.

The various defines for TCTL1, TMSK1, etc. can be found in the supplied header file hc I I .h.

All we have to do next is to initialize the interrupt vector, which is done in the file vectors.c. The main function needs to perform other initialization:

```
main()
        {
        TCTL l I= 0x3:
        TMSKI I= 0x4:
        asm("cli"); /* enable interrupt */
        /* just loop forever */
        while (I )

        }
```

main() simply turns on the appropriate control registers, enables the interrupt and finally just loops forever.

Volatile Type Qualifier

In C, you use the volatile type qualifier to tell the compiler that a data item may change in ways that are not apparent to the compiler. For example, memory mapped peripheral registers are good examples. Normally, the compiler may optimize certain memory accesses away. For example, if you are reading alocation immediately after writing to it, the compiler may reuse the temporary value in a register instead of reloading it.. For memory mapped registers, the effect is incorrect if the read is not done. Therefore, the peripheral registers must be accessed using volatile type qualifier, which is the case if you use the defines in the provided header file.

Debugging

What To Do If Your Program Does Not Work

You should examine the .mp 'map file (-m to the driver) to check that the memory addresses are reasonable and within the memory map of your system. For example. *when* using a single chip system and downloading using a program such as pcbug or dlm, you cannot have any data that is going into RAM (e.g., at address 0 and up). Also, the memory allocation routines uses memory just beyond that of the variable "__FreeList."[1] If you put code or data after this variable make sure that they will not be overwritten by the dynamic memory (memory allocation is done within blocks of memory created by the _NewHeap calls. Thcrc is a default one in the startup file).

It is also possible that your stack is overrunning your code or data. If that's the case. you must either relocate the stack at a different address and give it more space or change your program structure.

Liberal uses of printf and even primitive debuggers such as BUFFALO and can be used to pinpoint problem areas. You can also use the optional NoICE11 to debug RAM based code (which after debugged, you can put into ROM).

Also, BUFFALO overwrites location 0x4000. Make sure your program does not use that space, perhaps by using the linker memory range options.

1. The previous versions of the compilers use .map extension instead of .mp. However, since the compilers now support the P&E format .map file, the extension name has been changed.

Debugging Support

The compiler produces symbol table compatible with P&E debuggers. Before startintg debugging, please refer to the chapter "Troubleshooting: Frequently Asked Questions" 'on common problems that users have experienced. In addition to provide some source level information such as source line number and file name, and global symbol addresses for use with NoICE11. The compiler also provides:

1. A List of Local Variable Offsets

   Global and static variables are allocated from the data area and show up in the generated assembly code as symbolic labels in the data or bss sections. Thus they show up in assembly listings and linker map files and you can find out their physical addresses in the final program. In contrast, local variables are allocated on the stack and do not have fixed addresses. Rather, they arc referenced using the X index register plus an offset.

   For each function, the compiler generates a list of the mapping between a local variable and its assembly reference and inserts them as comments in the assembly file. For example, for this C function fragment:

   ```
   main( int argc, char *argv[])
               {
               int a, b, c;
               ...
   ```

   The compiler may generate this list:

   ```
   _main::          ; function entry point
   ; argv           -> 8,x
   ; argc           -> 6.x
   ; a              -> 2,x
   ```

2. Interspersed C and assembly output file.

   The compiler emits C source lines interspersed with the generated assembly code if the -l switch is specified. This switch also directs the driver not to delete the assembly file once the assembler is run.

1. See 'Troubleshooting: Frequently Asked Questions" on page 43.

3. The **linker** generated listing file (.lst, also controlled by the -m flag to the linker) contains all your assembly listing (.lis) files concatenated with the final addresses. The .map map file is a debug file compatible with P&E Microcomputer Inc. debug tool. You use the -g flag to generate the .map file. The listing file format is compatible with the TECI ICE emulator.