ME 6405

Introduction to Mechatronics

Fall 2004

Instructor: Professor Charles Ume

Lecture on Introl C compiler
Contents

• Overview of C compliers for HC11
• Overview of Introl CODE®
• Introl CODE Project
• Project Configuration
• Data Type Size Implemented by Introl C
• Variables Associated to Registers
• asm Pseudo Function
• Pointers
• Standard I/O
• Function Modifiers
• Interrupts
• Wrap up with an Example
Overview of C compilers for HC11

• C compiler: C language → assembly language
  – Offers a way to develop HC11 based system with C language.

• Commercial C compilers:
  – Dunfield Development Systems, Avocet, Cosmic, Fabius Software Systems, Introl CODE …..

• Freeware C compilers:
  – gcc11, ICC11, Interactive C, Small C, WCC11……
Overview of Introl CODE®

• **CODE -- COntroller Development Environment**
  
  – An integrated tool that helps develop programs for embedded systems using Motorola microcontrollers
  – Offers full ANSI compliance, library source
  – Including an assembler, a linker, and a debugger
  – Introl CODE supports all Motorola line (6805, 6808, 6809, 68HC11, 68HC12, 68HC16, and 68300 families).

• Also called Introl C complier, the version available in the mechatronics lab is: 4.00.1.12
Introl CODE® Project

• A CODE project is:
  – a collection of files and rules for performing operations on those files

• The files in a project can be:
  – source files, in either C or assembly language
  – project files, known as subprojects, that are built before the source files in a project are built
CODE Project Configuration

• We need to answer the following questions to configure a CODE project:
  – What is the target processor? (HC11)
  – What on chip resources does the processor have? (What variant: E9, how to initialize registers)
  – What kind of memory resources exist on the target and where are they? (memory configuration)
  – Is there any software running on the target? (like Buffalo)
  – How do I set up the processor's interrupt and exception vectors? (explain later)
  – Where is "Hello world" supposed to go? (Specify standard I/O)
CODE Project Configuration (Cont.)

• Specify processor family and its variant
• Configure environment
• Configure memory

• Refer to IntrolConfig.ppt (available on Mechatronic lab website) for detailed information on Introl C configuration
Size of Data Type Implemented in Introl C Compiler

<table>
<thead>
<tr>
<th>Type</th>
<th>#Bits</th>
<th>#Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>long double</td>
<td>64</td>
<td>8</td>
</tr>
</tbody>
</table>
Variables Associated to Registers

- Introl C associates the following variables to the HC11 registers. Include “hc11e9.h” to use the predefined variables;
- The size of the variables match the actual size of their corresponding HC11 registers;

```c
#include "hc11e9.h"  // Port A
#include "hc11e9.h"  // Parallel I/O Control
#include "hc11e9.h"  // Port C
#include "hc11e9.h"  // Port B
#include "hc11e9.h"  // Alternate Latched Port C
#include "hc11e9.h"  // Data Direction Port C
#include "hc11e9.h"  // Port D
#include "hc11e9.h"  // Data Direction Port D
#include "hc11e9.h"  // Input Port E
#include "hc11e9.h"  // Compare Force
#include "hc11e9.h"  // OC1 Action Mask
#include "hc11e9.h"  // OC1 Action Data
#include "hc11e9.h"  // Timer Counter
#include "hc11e9.h"  // Input Capture 1
#include "hc11e9.h"  // Input Capture 2
#include "hc11e9.h"  // Input Capture 3
#include "hc11e9.h"  // Output Compare 1
#include "hc11e9.h"  // Output Compare 2
#include "hc11e9.h"  // Output Compare 3
#include "hc11e9.h"  // Output Compare 4
#include "hc11e9.h"  // Input Capture 4/Output Compare 5
#include "hc11e9.h"  // Timer Control 1
#include "hc11e9.h"  // Timer Control 2
#include "hc11e9.h"  // Main Timer Interrupt Mask 1
#include "hc11e9.h"  // Main Timer Interrupt Flag 1
#include "hc11e9.h"  // Misc. Timer Interrupt Mask 2
#include "hc11e9.h"  // Misc. Timer Interrupt Flag 2
#include "hc11e9.h"  // Pulse Accumulator Control
#include "hc11e9.h"  // Pulse Accumulator Count
#include "hc11e9.h"  // SPI Control
#include "hc11e9.h"  // SPI Status
#include "hc11e9.h"  // SPI Data
#include "hc11e9.h"  // SCI Baud Rate Control
#include "hc11e9.h"  // SCI Control 1
#include "hc11e9.h"  // SCI Control 2
#include "hc11e9.h"  // SCI Status
#include "hc11e9.h"  // SCI Data
#include "hc11e9.h"  // A to D Control
#include "hc11e9.h"  // A to D Result 1
#include "hc11e9.h"  // A to D Result 2
#include "hc11e9.h"  // A to D Result 3
#include "hc11e9.h"  // A to D Result 4
#include "hc11e9.h"  // System Configuration Options
#include "hc11e9.h"  // Arm/Reset COP Timer Circuitry
#include "hc11e9.h"  // EEPROM Programming Control
#include "hc11e9.h"  // Highest Priority 1-bit and Misc.
#include "hc11e9.h"  // RAM/IO Mapping
#include "hc11e9.h"  // Factory Test Control
#include "hc11e9.h"  // COP, ROM, & EEPROM Enables
```
asm Pseudo Function

The Introl-C compiler recognizes a function named “asm”. The compiler interprets it as a request to insert a string directly into the assembler source file. The function is used with the following form:

\[
\text{asm(string)};
\]

Examples:

\[
\text{asm("	ORG $00D0");}
\]
\[
\text{asm("\tJMP Address");}
\]
\[
\text{asm("\tSWI");}
\]

It is also possible to access C functions and data from separate assembly language files, please refer to Calling Conventions in the online manual for detailed information.
• What is a Pointer?
  – A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

  – The operator & gives the “address of a variable”.
  – The operator * gives the “contents of an object pointed to by a pointer”.

  – To declare a pointer to a variable do:

    ```
    int *pointer
    ```
• A simple example
  – Assume variable x resides at memory location 2000, y at 3000, and ip at 5000

```c
int x = 1, y = 2;
int *ip;
ip = &x;
y = *ip;
x = ip;
*ip = 3;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Data stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2000</td>
</tr>
<tr>
<td>y</td>
<td>3000</td>
</tr>
<tr>
<td>ip</td>
<td>5000</td>
</tr>
</tbody>
</table>

The George W. Woodruff School of Mechanical Engineering
• A simple example
  - Assume variable x resides at memory location 2000, y at 3000, and ip at 5000

```c
int x = 1, y = 2;
int *ip;
ip = &x;
y = *ip;
x = ip;
*ip = 3;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Data stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2000</td>
</tr>
<tr>
<td>y</td>
<td>3000</td>
</tr>
<tr>
<td>ip</td>
<td>5000</td>
</tr>
</tbody>
</table>
• A simple example
  Assume variable $x$ resides at memory location 2000, $y$ at 3000, and $ip$ at 5000

```c
int x = 1, y = 2;
int *ip;
ip = &x;
y = *ip;
x = ip;
*ip = 3;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Data stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2000</td>
</tr>
<tr>
<td>y</td>
<td>3000</td>
</tr>
<tr>
<td>ip</td>
<td>5000</td>
</tr>
</tbody>
</table>
• A simple example
  – Assume variable x resides at memory location 2000, y at 3000, and ip at 5000

```
int x = 1, y = 2;
int *ip;
ip = &x;
y = *ip;
x = ip;
*ip = 3;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Data stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2000</td>
</tr>
<tr>
<td>y</td>
<td>3000</td>
</tr>
<tr>
<td>ip</td>
<td>5000</td>
</tr>
</tbody>
</table>
Pointer (cont.)

- **Note:** When a pointer is declared it does not point to anywhere. You must set it to point to somewhere before you use it.

- So below code may cause your program to crash:
  ```c
  int *ip;
  *ip = 100;
  ```

- Correct use:
  ```c
  int *ip;
  int x;
  ip = &x;
  *ip = 100;
  ```
Pointers (cont.)

- Example of pointers: Write a program to add odd numbers located in addresses $0000$ through $0020$
  ```c
  void main(void){
    int sum = 0;
    char* mempointer;
    mempointer = (char*) 0x0000;
    while(mempointer <= (char*) 0x0020){
      if(*mempointer & 0x01)
        sum = *mempointer + sum;
      mempointer++;
    }
  return;
  }
  ```

- In C programming, it is NOT recommended to directly assign an address value to a pointer like in the above example
- If you need to access certain memory location in C, you need to make sure that compiler is not using that part of memory for something else
Pointers and Functions

Example: if we want to write a function to swap variables around, the codes below on the left will not work (why?), but the one on the right will.

```c
void swap(int a, int b){
  int temp=a;
  a=b;
  b=temp;
  return;
}

void main(void){
  int a = 5, b=6;
  swap(a,b);
  return;
}
```

```c
void swap(int* a, int* b){
  int temp=*a;
  *a=*b;
  *b=temp;
  return;
}

void main(void){
  int a = 5, b = 6;
  swap(&a,&b);
  return;
}
```
Pointers and Arrays

- Pointers and arrays are closely related:
  ```
  int a[10], x;
  int *pa;
  pa = &a[0]; /* pa pointer to address of a[0] */
  x = *pa;
  ```

- However they are different:
  - A pointer is a variable. We can do `pa = a` and `pa++`
  - An array is not a variable. `a = pa` and `a++` are illegal

- When an array is passed to a function what is actually passed is its initial elements location in memory
  ```
  int MyArray[80];
  ArrayFunction(MyArray);  <->  int MyArray[80];
  ArrayFunction(&MyArray[0]);
  ```

Facts:
- `pa=&a[0]`;
- `pa+i` ≡ `&a[i]`
- `a[i]` ≡ `*(a+i)`
- `&a[i]` ≡ `a+i`
Pointers and Multidimensional Arrays

• A two dimensional array can be think of as a one dimensional array with each element being an array
  – E.g. int multi[2][4] can be visualized as a two element array, with each element being an array of four integers.
  – Since the name of an array is a pointer to the first array element, multi is a pointer to the first element, multi[0], in this case.
  – Since the element multi[0] is also an array (with array name multi[0]), multi[0] is also a pointer to the first element, multi[0][0] in this case.
  – While multi, multi[0] and &multi[0][0] all point to the same location, the sizes of the items they are pointing to are different.

• For any n-dimensional arrays:
  – The array name followed by n pairs of brackets evaluates as array data
  – The array name followed by fewer than n pairs of brackets evaluates as a pointer to an array element
Pointer Operations

- **Assignment:** e.g. `ptr=&x; ptr=array`
- **Dereferencing:** e.g. `*ptr`
- **Address of:** e.g. `&ptr`
- **Incrementing:** e.g. `ptr+=3`
- **Decrementing:** e.g. `Ptr--`
- **Differencing:** e.g. `ptr1–ptr2`
- **Comparison:** e.g. `ptr1<ptr2, ptr1==ptr2, etc.`
Pointers to Pointers

• Since a pointer is itself a variable, it is stored at a particular address, therefore you can create a pointer to a pointer

```
int myVar;
int *ptr = &myVar;
int **ptr_to_ptr = &ptr;
```
Pointers to Functions

• To declare a pointer to a function, use:
  – type (*ptr_to_func)(parameter_list):
    e.g. int (*ptf)(int x)

• Initializing and using a pointer to a fcn:
  float square(float x)
  float (*ptr)(float x)
  Float square(float x)
  { return x*x;}
  ...
  ptr=square;

• A function name without parentheses is a pointer (constant) to the function.
Standard I/O

- The ANSI functions printf() and scan() implemented in Introl C compiler assume an infinitely long string, it usually produce a large size machine code. Callers must be careful not to overflow the actual space; this is often impossible to assure.

- For example:
  - The following two programs are doing exactly the same thing: to output the string “hello” to the standard output device.
  - When compiled by Introl C complier:
    - Program 1 (puts) produces machine code size of 400 bytes
    - Program 2 (printf) produces machine code size of 6,300 bytes

```
Program 1
#include <stdio.h>
void main(void)
{
    puts("hello");
    asm("\tSWI");
}

Program 2
#include <stdio.h>
void main(void)
{
    printf("hello");
    asm("\tSWI");
}
```
Function Modifiers

- **__protected (double underscore)**
  - Causes a function to be generated that will run with all interrupts disabled.
  - The current interrupt state is saved when the function is entered, the interrupts are disabled and the body of the function is executed. The previous interrupt state is restored when the function returns. The disabling of interrupts only applies to those interrupts that can be disabled under software control.

- Example:
  ```c
  __protected void SampleFunc(void)
  {
  ...
  }
  ```
Function Modifiers (Cont.)

• __interrupt
  – Specifies a function to be generated that can be used directly as an interrupt handler (interrupt service routine).
  – A function declared with the __interrupt modifier must not be called directly from C since the function performs a return from interrupt when it is finished.
  – The name of an interrupt function can be used in the Vectors tab of the project editor to define an exception handler. An interrupt function should return void and have a void parameter list.

• Example:
  
  __interrupt void handler(void)
  {
    ...
  }

  The George W. Woodruff School of Mechanical Engineering
Interrupt

First, do not turn on the "setup vectors" and "setup interrupts" when you configure your project. The following is an example of using interrupt when programming with Introl C.

```c
/* Put "Tick" on the screen every second using timer overflows*/

/* Include standard output and hc11 register definitions */
#include <stdio.h>
#include <hc11e9.h>
int counter; /* global variable */

/* The function modifier "__interrupt" is required for
a interrupt routine function. Please be aware that
there are two underscores in the modifier */

__interrupt void Time(void) {
    counter++; /* Increment the global variable counter by 1 */
    if(counter == 30) /* An over flow occurs every 33 ms so 30 times a sec */
        { 
            puts("Tick
"); /* Put "Tick" on the screen followed by a newline */
            counter = 0; /* reset the counter */
        }
    _H11TFLG2 = 0x80; /* Clear a the interrupt flag before exiting the
function*/
}

/* Notes: Interrupt function takes no argument and returns nothing. */
/* The counter cannot be declared in the interrupt function because it
would reset it's value every time the interrupt is run. */
/* Also to clear the flag in TFLG2 you have write a one to that bit. 
This goes for any other register that says to "write bits to clear" */

/* the main function that setup the interrupts*/

void main(void)
{
    asm("SEI"); /* A couple of things. /* SEI masks maskable Interrupts */
    /* It is recommended to do this before initializing any interrupts */
    _H11TFLG2 = 0x80; /* Clear the interrupt flag for Timer overflow */
    _H11TMSK2 = 0x80; /* Turn on the Timer overflow interrupt itself */
    counter = 0; /* Initialize global counter variable */
    asm("CLI"); /* Enables maskable interrupts */
    /* Your interrupts should start at this point */
    while(1); /* infinite loop */
    return;
}

/* Setup the interrupt service routine starting address */

asm("ORG $00D0"); /* Timer overflow vector is at $00D0 */
asm("JMP Time"); /* Write a jump to the interrupt function in the vector */

End of program */
```
Wrap Up with Another Example

- LED lab with C code:

```
#include <hc11e9.h>
void delay(unsigned long seconds) { /* function to delay 1 second*/
    unsigned long clocks = seconds*12700;
    while(clocks > 0) {clocks = clocks-1;}
    return;
}

void main(void) {
    short i = 0xFC; /* FC = 1111 1100 */
    _H11DDRD = 0xFF; /* Initialize port D. */
    while(1) { /* Loop to show binary increase. */
        _H11PORTD = i;
        i = i - 4;
        delay(1);
    }
    return;
}
```