CHAPTER 3

Using the IDE

This chapter is a snapshot of the IDE's online help. While we try to maintain this as up to date as the online version, the online help always contains the latest information and should be used as the primary reference for the IDE.

The ICC 11 Integrated Development Environment (ICC 11 IDE) is a Windows program that includes a Project Builder, a text editor. a terminal program, plus integration with the command line compiler tools. The tight integration of tools allows fast edit-compile-download development cycle of HC 11 programs.

The menus are context sensitive in that operations that are not valid are disabled. For example, the compiler menu is only enabled if the active window is an editor window. The button bar buttons are always enabled but context sensitive buttons would simply print an error on the status window if they are not being used in the correct context.

Options

Before you use the IDE, you probably want to set options for the different components of the IDE, especially for the compiler setup.

Compiler Option

This option tab controls the operations of the compiler. There are only a few items that must be setup for your target before you can run programs. The details are explained in Chapter 4. Note that hex addresses must be specified using C notation, preceded by Ox, e.g. 0x8000. Thk easiest way is to click on the "Setup Wizard" button on the Option->Compiler->Linker tab. This contains a list of configurations that you can select, save, and modify. The important options are:

1. Specifying address of the text or code section. This is either ROM or RAM address depending on your target system.
2. Specifying address of the data section. If you leave the option blank, then the data section simply follows the text section, which is the case if you are testing on a RAM based system where both the code and data go into RAM. Otherwise, specify an address for the data section which usually is the "bottom" of your RAM.
3. Specify your initial stack pointer value. Usually this is the top of your RAM.
4. Specify heap size. Heap is used only for dynamic memory allocation routines such as malloc() and free(). If you do not plan to use any heap space, leave it as 0.

Specify the startup file name. The default is crt11.o. You may use different startup files for different target setup. You should be able to run programs after making these changes. If you are testinp with printf, then you may also need to modify the putchar() function in the library.

The compiler options are stored in a Windows .ini file or in a project file if a project is active.

Target Option

This option tab controls the operation of the Target Window, an ANSI compatible Terminal Emulator. Additionally. it controls:

1. COM port parameters such as COM port number and the baud'rate. This does not affect the bootstrap programming baud rate.
2. The font for the terminal emulator.

For internal EEPROM, you can set the board echo mode to either normal or MIT Miniboard style. You can also optionally set Bulk Erase mode. Otherwise, internal EEPROM bytes are erased a byte at a time if necessary. Bulk Erase also causes the config register to be programmed to 0xF5, a typical value for the 'E2 HC 11.

You can optionally set the config register to a particular value as part of the bootload programming process by modifying the config register edit box.

Bootload programming is always done at 1200 baud, assuming the target having a 8MHz clock.

Editor Option

Editor options include:

1. Autoindent mode.
2. Editor font.
3. Print options including whether to print line number and whether to include a print 'margin on the left.
4. Syntax highlighting mode, Note that changing this option will not cause the current display to switch. It only affects new displays (e.g. moving around in the editor window).

Project Builder

Typically an executable (.s 19 file) is made up of multiple source files (.c and .s files). The Project Builder manages this kind of multi-file project automatically, including generating header file dependencies.

You may put project files in any directories but it is usually the best to have a separate directory (or directories) to put your source and header files. The name you give for the project file (<file>.ipj) will bc used as the name of the executable output (<file>.s19). When you create or open a project. the Project Builder window appears.

Project Window

The project window lists all the files that are in the project. When you double click on a file on the list. an editor window opens with that file. Use the "Add" button to add files to the project. You may select multiple files at once with the file selector dialog box. Use the "Remove" button to remove any selected files from your project file list. You can use the standard Windows multiple selection methods: Control-mouse-click to select another file, and Shift-mouse-click to select the range of files.

You can specify a private makefile to be included at the end of the project makefile. This is useful if you want to specify your own rules or include non-C or non-assembly source files. You can edit the file name manually or use the "Add" button to bring up **a file open dialog box.**

Building A Project

When you request building a project, the Project Builder generates the dependency information if necessary, creates a makefile from the project file list and the current compiler options, and then invokes the imake utility to "make" the project.

Note: C header file dependencies are generated only for non-system header tiles, i.e. header files that are included via double quote syntax and not <>. Also, *C* header file dependency generation understands the C preprocessor directives and compiler options, but the assembly header file generation does not understand assembler directives such as .if.

Terminal Window

This is where you communicate with your target system using the builtin terminal emulator. The terminal emulator emulates an ANSI terminal (similar to the DOS ANSI.SYS driver) so you can have your target output cursor control directives. File capture is supported.

ASCII Download

Downloading support includes simple ASCII downloading. Typically this is for a target running a monitor such as BUFFALO. You would enter download mode (for BUFFALO, type "load t") and use this command to transfer a .s 19 file lo your target.

Bootstrap Mode Programming

You can also use the terminal component to program your target using the bootstrap programming mode of the HC 11. Refer to your target system reference on how to set the system to run in this mode. By using the bootstrap programming mode, you can program an HC I 1 target without a monitor running on the target. Examples of some commercial HC 11 systems that can utilize use this mode are:

1. MIT Miniboard, TechArts Adapt- 11 Starter Kit and other 'E2 single chip based system. The program memory is just the internal 2K of EEPROM.
2. MIT Handy Boards with battery backed external RAM.
3. TechArts Expanded Microcontroller Kits with external X68C75 EEPROM and external normal EEPROM and RAM.

The main difference between X68C75 EEPROM and normal external EEPROM is that the X68C75 requires the Software Data Protection (SDP) register to be disabled prior to programming. If you want to use the SDP facility of the X68C75, then you must re-enable this register in your application code.

The bootstrap programmer only programs one type of memory for each download. as specified in the Option->Target option tab. If you have a system with multiple memory types (for example, a TechArts expanded kit with both external RAM and X68C75 EEPROM), then you will have to download the pieces separately.

Bootstrap programming is a two step process: first a small bootload program is loaded into the internal RAM at 1200 baud, then the application program is loaded by the bootload program at 9600 baud. Currently it only supports bootstrap mode at 1200 baud, thus assuming an 8MHz target. It supports fixed length bootload code such as for the A series and variable length bootload code such as for 'EO.

Note that the HC 11 variants that accept variable length bootload code such as the 'E0 have a very tight timing constraint in they will assume the bootload code is loaded when a character does not show up in 4 character time. Due to Windows multi-tasking capability, sometimes it is possible that the initial bootstrap is not successful. Simply reset your board and try again if this is the case. If the problem persists, try to disable background tasks, especially virtual DOS boxes in your Windows environment.

The IDE verifies each byte as it is written so there is no need for a separate verify function.

Editor

The editor component understands C syntax, can handle larger than 64K byte files and contains most of the usual edit control.

Editor Key

Ctrl+Left	one word left
Ctrl+Right	one word right
Ctrl+Up	to the beginning of the paragraph
Ctrl+Down	to the end of the line
Home	to the beginning of the line

End	to the end of the line
Page Up	page up
Page Down	page down
Ctrl+Home	to the beginning of the text
Ctrl+End	to the end of the text
Ctrl+PgUp	to the beginning of the visible page
Ctrl+PgDn	to the end of the visible page
Shift	When combined with previous keys (including mouse click), expands the current selection
Insert	toggles insert / overwrite mode
Delete	deletes one character to the right or the selected text
BkSp	deletes one character to the left or the selected text

Miscellaneous Features

The editor supports the Windows standard cut / copy / paste, search and replace, up to 5 bookmarks. multiple levels of undo and redo, automatic conversion of Unix ASCII file format (single linefeed for line termination) to MS-DOS ASCII format (carriage return followed by linefeed for line termination), goto line command etc. You can also compile or print an opened file.

Double clicking an error message with the format:

```
!E <filename>(<line #>): . . .
```

brings up an editor with the file loaded and the cursor pointing at the offending line.

Double clicking a tile name in the Project Window's file list also brings up the file in an editor window.

CHAPTER 4

Customizing The Compiler

Since the ICC11 compiler does not assume any particular target system configuration, you must first customize the compiler before you can use it. The steps are as follows:

1. Customize the linker by specifying the locations where you want to put the code (text) and data sections (e.g. where your system's ROM and RAM are), **how big the heap is, and** where the stack pointer is. This can be done by invoking the "Setup Wizard" in the IDE's Option->Compiler->Linker tab and choosing a configuration that most closely matches yours.
2. Additional complications exist for the HC 11'F1 since the memory selection logic may need to be setup. This can be done in _HC11Setup() with the caveat that the first instruction of crtl 1.s initializes the stack pointer and the stack region may not be visible yet! In this case, you'll have to set **the stack** pointer to the top of the internal RAM. calls _HC 1 1 Setup, and then set the stack pointer to the normal value afterward.
3. [optional] If your target system needs to perform special functions (such as remapping the IO registers) in the first 64 cycles, you can put the code in the function _HC I I Setup() and put in the library 1 ibc 11 . a.
4. [optional] Change the header file `hc11.h` where it specifies the base address of the internal IO registers and the register names.
5. [optional] Create character *IO* (input / output) routines.

6. [optional] Setup the interrupt vectors.

Runtime Startup File

The linker automatically links in the start-up object file `crt11.o` before user specified files'[1] to generate the executable (it also includes the object file `end11.o` and the library file `libc11.a` after the user input files, but `end11.o` should not need any changes). Thus the entry point of your program is the beginning of `crt11.o`, defined by the symbol __start. You can specify a different startup file by using the linker -u flag, or IDE's Option->Compiler->Linker->"Startup File" edit box. You may want to use different startup files if you have multiple target boards.

The start-up code performs the following tasks:

1. Initializes the bss section[2] to zeroes.
2. Initializes the stack pointer. The value of the stack pointer is specified as a linker symbol.
3. Copies the initialized data from "idata" section to "data" section.
4. If requested, initializes dynamic memory allocation space after the BSS section.
5. Jumps to the user main() routine.

The stack pointer is set up via this instruction:

```
        lds     #init_sp
```

You should define the value of init_sp in your linker command option or in IDE's Option->Compiler->Linker:

```
        -dinit_sp:0x7FFF
```

A common practice is to put the global data at the bottom of your RAM, and your stack at the top of your RAM. Since the stack grows downward. this convention gives the most space to the stack.

1. Specifying the -R switch to the linker disables automatic linking of these files with your program. This is useful for creating standalone library ROMs.
2. The bss section is where the uninitialized global C variables reside. By ANSI C definition, they will get initialized to zeroes at program startup time.

Since different target configurations may use different chip selects or have different hardware setup, you can have multiple start-up files in addition to the default `crt11.o`. You use the -u <startup file> option or the IDE Option->Compiler->Linker->"Startup File" to select a non-default start-up file.

`Crt11.o` calls the function _HC11Setup() after the stack pointer is set up. The default version in the library `libc11.a` does nothing but simply returns. If your system needs initialization code to be performed during the first 64 cycles after reset, then you can replace this library function with your own.

Dynamic memory allocation heap is controlled by the symbol heap_size. If it is non-zero, then `crt11.o` calls the heap setup routine _NewHeap to initialize that amount of space after the BSS section for use by the memory allocation functions. You should define the value of heap-size in your linker command option or in the IDE's Option->Compiler->Linker:

```
-dheap_size:2000
```

Note that the call in the start-up file is there and thus the code for _NewHeap will still be linked into your program and take up code space even if you define heap-size to be zero. To reclaim the space. you must manually delete the call to _NewHeap in the start-up file and recompile it. This is especially useful for limited memory targets such as the HC11'E2 with just the internal 2K EEPROM for program storage.

If you modify the start-up .s file, generate a new .o and place it into the library directory. For example:

```
icc11 -c crts11.s
copy crts11.o c:\icc\lib
```

Link Addresses

The compiler allows you to place sections[1] at different memory addresses. You can specify where the text section starts by giving the -b (base) switch to the linker. The default is 0. For example, specifying

```
-btext:0x8000
```

1. By default. the compiler puts code in the "text" section. and global data in the "data" section. You can also use absolute section: sections with explicit addresses in them. See the chapter on Linker for more details on sections.

to the linker means "place the text section at address 0x8000." You can specify the data section start address like this

```
-bdata:0x2000
```

This puts the data section at address 0x2000. The complete syntax is very flexible, allowing you to specify memory ranges, which may be useful if you have discontiguous memory devices. Note that the linker does not give you a warning if the sections overlap, and you must ensure that this situation does not occur. Similarly, the bss section normally starts at the end of the data section, but you may specify a starting address for the bss section with the -b switch to the linker.

If you are using the IDE, you can specify the section addresses by choosing "Option->Compiler->Linker." If you arc not using the IDE. you can specify the -b switches in the **ICC11_LINKER_OPTS**, usually set in the `icc11set.bat` file.

in a ROM based system, you probably would give text and data sections separate addresses, putting them into ROM and RAM respectively. In a RAM based system, you probably would just specify the address for the text section, and let the data section begins where the text section ends. This is the default behavior if you don't specify a starting address for the data section.

_HC11Setup()

If you need to perform some target specific configuration in the early part of your program after reset, you may do so by putting the code in the function _HC11Setup() and replacing it into the library. The assembly name is __HC11Setup (two underscores) and the symbol must be made global:

```
__HC11Setup::
        ; whatever
        rts
```

The assembly file name must be named setup.s. After assembling it, you replace it into the library by using the librarian:

```
ilib -a libc11.a setup.o
```

HC11.H

The file HC11.h contains the addresses of the internal registers. If you wish to use the file, you must adjust the value IO-BASE if it is not at the default address of 0x1000. *You* may also need to add more entries for other variants of the target processors. It also defines enum for baud rate for a system with 8 MHz system clock.

Character IO Routine

If you wish to use any of the library output functions such as printf(), you must implement the function putchar() which writes out one character. The supplied routines putchar() and getchar() are in the file iochar.o. They use the SCI port in polling mode. If your IO system uses some other method (e.g. you want to ouput to a LCD device) or if the IO base is not at the default address, you will need to make the modifications and then compile iochar.c and replace them into the library:

```
ilib -a libc11.a iochar.o
```

Call the function setbaud() if your application needs to set up the SCI. This is usually the case unless your program is running under something like BUFFALO where it initializes the SCI port:

```
setbaud(BAUD9600)
```

This function initializes the SCI to the specified baud rate and enables the transmitter and receiver functions of the SCI. The baud rate is specified as a C enum which is defined in hc11.h also. It assumes the default system clock so if your system is different, you will need to modify hc11.h and recompile serial.c in your library source. If your IO registers are not at the default locations, you will need to modify hc11.h and recompile serial.c:

```
ilib -a libc11.a serial.o
```

Interrupt Vectors

The file vectrs11.c in the example directory contains the table of interrupt vectors. If you are putting your code in a ROM-based system (i.e. running without a monitor on the target), then you will need to use this file to setup the interrupt vectors. The file uses a C pragma to force loading of the vectors at the vector address. If your system moves the vectors somewhere else, you should modify this address. To use this file, you either create a multiple file project and include vectrsll .c as one of the project files (the IDE makes it easy to set up multiple file project), or you can simply "#include" this file at the bottom of you main program if you just want to use a single source file and not wish to set up multiple file project.

Each entry in the table is simply a function pointer. Therefore to put in your own entries, declare your interrupt handler functions in vectrsll .c and then put the names of the functions in the table entries, replacing whatever dummy entries that are in there. The file vec trsll .c contains comments and examples on how to do this.

Pseudo Vectors

If you are using a monitor such as BUFFALO or NoICE11, then your code is in RAM and these monitors usually provide a set of pseudo vectors mirroring the real vectors. For example, BUFFALO uses pseudo vectors at 0xC4 to 0xFF. Unlike the real interrupt vector entry, each BUFFALO's psuedo vector entry is 3 bytes long, the first byte being the instruction "JMP" and the last two bytes being the address of the interrupt routine for that interrupt. Since they are in RAM, you should modify them at runtime, typically early in your main() function. For example,

```
void toc5_handler();
main()
        {
        ....
        *(unsigned char *)0xd3 = 0x7E; /* 7E is "jmp" */
        *(void (**)())0xd4 = toc5_handler;
```

This will setup toc5_handler to handle the Timer Output Compare 5 interrupts under BUFFALO. Normally you would need to use the "#pragma interrupt-handler ..." statement to declare a function as an interrupt handler. See "Interrupt Routines" on page 36.

Examples of Customizing of ICC11 System

The IDE's Option->Compiler->Linker->"Setup Wizard" contains setup information for many of the popular HC 11 boards. Below is a sample:

Generic Single Chip Mode System

Since the HC11 has on chip ROM, RAM, and EEPROM. a minimal system consists of just the chip running in single chip mode. In this example, we will assume you are using an HC811E2. The HC 11 E2 has:

- 2K bytes of EEPROM at 0xF800, relocatable to the top of any 4K memory page.
- 256 bytes of RAM starting 0x0000.

In this example, we will put the code in the EEPROM, data and stack in RAM. We will need to compile vectors.c:

```
cd \icc I l\examples
icc I I -c vectors.c
copy vectors.o <my prog dir>
```

Next we change the **ICC11_LINKER_OPTS** environment variable to the fol-'lowing: (all in one line)

```
set ICC11_LINKER_OPTS=-btext:0xF800 -bdata:OxO -dinit_sp:OxFF -
dheap_size:0
```

If you plan to use other interrupts, you will need to add them to the interrupt-vectors entries. The initial stack pointer is set to 0xFF, the top of internal RAM.

Since there are only 256 bytes of RAM for both the data and stack, the program must not use too much stack space (e.g., allocating large amount of local variables or performing many levels of function calls) or data space (e.g., global variables). We probably do not want to do dynamic memory allocation in this setup due to , small amount of RAM; therefore we the heap-size symbol is set to 0.

Note that if you use a program to download the internal EEPROM, be aware that it cannot download data to RAM. Therefore you should not have any initialized global data.

Finally compile your fite (say foo.c) and link with vectors.o:

```
icc 11 foo.c vectors.0
```

Generic Expanded Mode System with BUFFALO

In this example, we will assume you have an HCI 1 system with 32K bytes of external RAM starting at address 0x0000 and internal BUFFALO ROM. Program download and execution is done through BUFFALO. We will not be using vectors.c in this case since BUFFALO controls the interrupt vectors.

Since the on-chip RAM and registers have priority over the external RAM, we will not be able to access the external RAM at addresses where they conflict with the internal resources (e.g., 256 bytes at address 0x0, and 64 bytes at Ox 1000). We will therefore set our code section to start at Ox 1800, with the data and bss sections following the code section, and the stack starting at the end of external RAM at 0x7FFF and growing downward. First change the definition for the environment variable **ICC11_LINKER_OPTS**. Since BUFFALO overwrites location 0x4000, that location should not be used:

```
set ICC I 1_LINKER_OPTS=-
btext:0x1800.0x3FFF:0x4002.0x7FFF -dinit_sp:0x7FFF -
dheap_size:0
```

Now we can compile and download programs with BUFFALO.