

Troubleshooting: Frequently Asked Questions

Sometimes your programs, even “simple” programs, may not work for some “inexplicable” reason. After thoroughly checking your code for programming errors, and before labeling it “a bug in the compiler” (as it may turn out to be on rare occasions - contact us for a fix!), here are several things to check first. (Other chapters in this manual contain more detailed explanations on how to perform the functions mentioned.) Note that our website <http://www.imagecraft.com> contains an up-to-date version of this chapter.

“Hello World” does not work

- **Is the baud rate set correctly?** BUFFALO sets the baud rate to the default 9600 baud, but if you are running a stand-alone program, you’d have to set the baud rate yourself.
- The default `putchar` function uses the **SCI** port. Is your RS-232 output on the microcontroller system coming from the **SCI** port? In particular, the Motorola **HC11** EVB boards use the **ACIA** chip instead of the **SCI** port. A solution for the Motorola EVB is to write the `putchar` function to use the BUFFALO character output routine instead.

Program hangs after finishing

- If your `main()` function “returns,” the transfer of control goes back to the code in the start-up file `cr t 11 .s`, and the default behavior is to execute an infinite loop. An alternative behavior, especially when executing under a monitor such as `BUFFALO`, is to return control to the monitor program. Note that under a monitor, a simple `“rts”` in `cr t 11 .s` after the call to `main` would not suffice since the start-up code loads a new value to the stack pointer as the first instruction, which is likely different from the one that the monitor was using. Thus when an `rts` is executed, it would not have the correct return address on the stack. To get the desired behavior, either replace the infinite loop with the `“swi”` instruction which would cause an interrupt trap back to `BUFFALO` or delete the `“lrs”` instruction.

Interrupt handlers do not work

- Did you enable interrupt via the `INTR_ON()` macro or the `“cli”` instruction?
- If you are using a C function as an interrupt handler, you must use the `interrupt-handler pragma` before you define the interrupt handler function.
- You need to set up interrupt vectors: stand-alone programs need to use the `vectors.c` file. Programs running under `BUFFALO` and `NoICE11` need to use the pseudo vectors in RAM. Note that each entry in `BUFFALO` pseudo-vector entry is 3 bytes long, please refer to `BUFFALO`'s documentation for information. Entries in `NoICE11`'s pseudo vectors are two bytes each.
- In order to use interrupts on some of the hardware subsystems, you must turn on the interrupt enable bits. For example, the Timer Output Compare enable bits are in the `TMSK 1` register. In most cases, you must also clear the interrupted flag by writing a `“1”` into the flag register in your interrupt handler. For example, the Timer Output Compare flag bits are in the `TFLG 1` register.

Printf does not print floating point number

- You must explicitly link in the `libfp.a` library. `“#include <stdio.h>”` in your source code is not sufficient.
- This `printf` is only a subset. In particular, precision and field width modifiers are not supported.

Programs do not work

- Is the COP Watchdog timer enabled accidentally? The Watchdog timer would interrupt and reset your target if enabled “on” accidentally. Refer to the HC11 reference manual for details. Basically, ensure that the NOCOP bit is “1” in the CONFIG register (i.e. the 3rd bit of 0x103F is 1).
- Program areas should not overlap. Check your .mp map files! In particular, setting data and text to the same address would cause problems.
- If your program is a stand-alone program, then it needs a reset vector!
- BUFFALO has a bad habit of writing to location 0x4000 due to an artifact of the Motorola EVB. You can either delete the offending code in BUFFALO and burn a new BUFFALO ROM, or use the linker address range to skip over the 0x4000 location.

Did you accidentally check the “No Startup File” checkbox in Option->Compiler->Linker? This option is useful only for building stand-alone modules such as ROM library code. **Bootstrap Mode Downloading does not work**

- Is the HC11 running with a 8Mhz clock? Does the target board support bootstrap mode?
- If you are using one of Technological Arts Adapt-11 boards, make sure that the “write enable” switch is off every time you power on or reset the board. Also choose the right type of memory device you are downloading in the bootstrap mode option.

Assembler cannot assemble assembler modules

- The ImageCraft assembler uses different syntax than Motorola freeware assembler. In particular, the bclr/bset/brclr/brset syntax is different.
- Direct page reference must be preceded with the ‘*’ character.

The purpose of a linker is to combine object modules and libraries together to form an executable image. In the case of ICC 11, the default executable image is in the form of a Motorola S record file, which is an ASCII file format for encoding program and data files. Most HC 11 monitors, EPROM burners, etc. take S record files as their input. You may also generate the executable image in Intel hex format by specifying a command line option to the linker.

Program Areas or Sections

Sections

Logically, an object and an executable file have 3 sections: the executable instructions are in the code (or text) section, initialized global or static data are in the data section, and uninitialized global or static data are in the bss section. Since there is no operating system or memory management hardware enforcing these restrictions on HC 11 systems, you may put code and data anywhere you choose. During runtime, a program has access to 2 other sections, the stack section for local variables and runtime control, and the heap for dynamic memory allocation.

Text

Text area is where your code goes. Typically this is in read only storage such as EPROM. During program development, it is typically placed in RAM for fast download. The power of a relocatable compiler system such as ICC1 is that your program does not need to be recompiled when you change the placement of your areas. Your program needs only to be relinked for it to work correctly (unless of course you use hard coded references in assembly routines).

**Data and Idata:
Initialized Global**

Initialized global variables, for example

```
int abc = 1;
```

is located in the data area. In order to initialize the data section with the correct values, the initialized values of the data area are actually kept in the "idata" section. At program startup time, the start-up routine copies the idata area to the data area. The idata area is located after the text area, so in a ROM based system, it will also be located in ROM and the content will survive power off and reset. 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 discontinuous 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.

BSS Area

Global variables that are not explicitly initialized, for example,

```
int def;
```

have zeroes as the initial values by ANSI C rule. They are placed in the BSS area¹. The startup code initializes the entire bss section before calling main().

Absolute Section

Sometimes it is convenient to put objects in absolute address. For example, for interrupt vectors. In C, you can use the the following pragma:

```
#pragma abs_address:<address>  
#pragma end_abs_address
```

1. BSS stands for Block Starts Section, a term carried over from the dawn of C-age.

example:

```
#pragma abs_address:0xFFD6 /* interrupt vectors for hc 11*/
```

The “abs_address” specifies that all items following are to be put into memory starting at the specified address. **You** may use additional “abs_address” pragmas if you need to put the additional items at specific locations. An “end_abs_address” pragma or end of file condition reverts to normal behavior:

For the “abs_address” pragma, the following assembly directives are generated:

```
.area    memory(abs)
.org     <address>
```

The “(abs)” after the section name “memory” specifies to the assembler and linker that this is an absolute section and does not need a linker address at link time.

Note that this pragma does not affect the placement of global or static data that are not explicitly initialized since they always go into the bss section.

Symbol

The assembler treats an undefined name as an external reference. For example, if the file contains:

```
ldd     #__heap_start
```

and there is no definition of `__heap_start` in the file, the assembler puts out an external reference of that symbol in the object file. When the linker combines the object files together, it will search all the files to find a definition of that symbol. Typically the symbol would be found in another object file. For example, the symbol may be an external variable and another file would have a definition of it.

In addition, the linker allows symbols to be defined at link command line. For example, if the symbol `init_sp` has the value `0x7FFF`, then you can use the command:

```
icc I1-dinit_sp:0x7FFF file.0
or
ilink -dinit_sp:0x7FFF file.0
```

In addition to using a numeric address, you may also use the value defined by a linker symbol that is previously defined in the linker command options. For example,

```
ilink-d__regs:0x 1000 -bdata:__regs
```

Assigns the value 0x 1000 to __regs and puts the data section at that address.

Link Address

The complete syntax for address specification is:

```
-b<name>:<range>[:<range>]  
<range> is <begin_address>[.<end_address>]
```

For example, if your system ROM space is at 0x0 to 0x 1FFF, and 0x4000 to 0x5FFF, you would write

```
-btext:0x0.0x1FFF:0x4000.0x5FFF
```

This allows you to maximize the use of discontinuous memory space. The maximum address is 0xFFFFF for ICC 16, and 0xFFFF for ICC 11 and ICC 12.

Default Address

If you do not specify an address for a section, the linker uses the ending address of the section preceding the section as its start address. This ordering is defined simply by the appearances of the section labels in an object file. Hence a simple way to enforce section ordering is to define the sections in the startup file. The default startup file specifies that the data section starts right after the code section using this method.

Introduction

The compiler generates assembler code which is then processed by the assembler. The assembler generates a relocatable object file from the input. You may also write assembler routines and link them into your C program. This chapter describes the format of the assembly language accepted by the assembler. This format is different from the Motorola Freeware Assembler. The `ias1lcvt` program translates a file with Motorola syntax to `ias6811` syntax.

Relocatable Sections

Assembly code is grouped into relocatable or absolute sections. The linker combines together sections of the same names from all the object modules. At link time, you specify the start address of each relocatable section and the linker adjusts symbol references to their final addresses. This process is known as relocation.

Assembly Source Code Format**Notation**

A <name> is a sequence of 32 or less characters consisting of alphabets, digits, dots (.), dollars (\$), or underscores (_). A name must not start with a digit.

A <number> is a sequence of digits in C format: a 0x prefix signifies a hexadecimal number, a 0 prefix signifies an octal number and no prefix signifies a decimal number. In addition, 0b signifies a binary number. You may also indicate hex number by using the \$ prefix.

An <escape sequence> is C style \n, \t, etc. plus \0xxx octal constants. \e refers to the escape character.

A <string> is a C string: a sequence of characters enclosed by double quotes (""). A double quote within the string must be prefixed by the escape character backslash (\).

An <exp> is a relocatable expression. It is either:

1. a term, i.e., a dot (which denotes the current program counter value ¹), a number, an escape sequence, a name, or
2. an expression enclosed with '(' and ')', or
3. two expressions joined with a binary operator. These binary operators, with the same meanings as in C, are accepted:

>>	<<	+	*	/	%
&		^			

1. Note that some HC 11 assemblers also accept a '*' to denote current program counter value. That does not work with this assembler

or

4. a unary prefix operator applied to an expression:
- > upper byte of an expression
 - < lower byte of an expression
 - .x character "x"
 - "ab" double byte value of characters "a" and "b"
- and the following C style unary operators:
- +

Unlike C, all operators have the same precedence so you may have to insert parentheses to group the expressions. Also note that only one relocatable symbol may appear in an expression.

Direct Page Reference

Some instructions take a direct page reference as an operand. That is, the address of the operand is in the first 256 bytes of memory. You cannot specify a direct page variable in C, but you can do so in assembly by prefixing a variable or a direct address with a '*' in the references:

```
bset    *_foo,#0x23
bclr   *0x20,#0x32
```

You must ensure that a direct page variable is defined within the first 256 bytes. Otherwise, you will get an error when you link the program. You may do so by putting such variables in an absolute area:

```
.area   memory(abs)
.org 0
_foo:: byte 1
```

Format

An assembly file consists of lines of assembly text in the following format. Lines greater than 128 characters are truncated:

```
[ label: ] operation [ operand ]
```

In addition, comments can be introduced anywhere on the line with the ';' character. All characters remaining in the line after the comment character are ignored.

A label defines a relocatable symbol name. Its value is the program counter value at the point where the label appears in the final linked executable. Zero, one, or more labels may exist on a source line. A label must end with a single colon ':', or two colons '::', the latter case signifying that the label is a global symbol (that is, one that can be referenced from another object module).

An operation is either an assembler directive or an HC 11 opcode.

Assembler Directives

Directives are operations that do not generate code but affect the assembler in certain ways. The assembler accepts the following directives:

.text specifies that the following data and instructions belong to the text section.

.data specifies that the following data and instructions belong to the data section. If you create any data items in the data section in an assembler module, you must define the same values in the `idata` section immediately following it. At program startup time, the `idata` section is copied to the data section and the sizes of the two sections must match. In fact, it is better to reserve the space in the data area and define the values in the corresponding `idata` area. For example:

```
.data
_mystuff::
        .blkb 5
.area idata
        .byte 1, 2, 3, 4, 5 ; _mystuff gets these at startup
```

.area <name> specifies that the following data and instructions belong to a section with the given name. `.text` is a synonym for `.area text` and `.data` is a synonym for `.area data`. The compiler only uses the `text`, `data` and `bss` sections.

`<name>` may optionally be followed by the attribute `"(abs)"`, signifying that this area is an absolute section and can contain `.org` directives.

.org <exp> change the program counter to the address specified. This directive is valid only within an absolute area.

.byte <exp>[,<exp.>]* (or .db . . .) allocates bytes and initializes them with the value(s) specified. For example, this allocates 3 bytes with the values 1, 2, and 3.

```
.byte 1,2,3
```

.word <exp>[,<exp.>]* (or .dw . . .) allocates words and initializes them with the value(s) specified. For example, this allocates 3 words with the values 1, 2, and 3.

```
.word 1,2,3
```

.blkb <number> reserves *number* bytes of storage without initializing their contents.

.blkw <number> reserves *number* words of storage without initializing their contents.

.ascii <string> allocates a block of storage large enough to hold the string and initializes it with the string.

.asciz <string> allocates a block of storage large enough to hold the string plus 1, and initializes it with the string followed by a terminating null.

.even forces the current program counter to be even.

.odd forces the current program counter to be odd.

.globl <name>[,<name>]* declares that name(s) are global symbol(s) and can be referenced outside of this object module. This has the same effect as defining the label with 2 colons following it. That is

```
_foo:  
    ; some asm code  
.globl _foo
```

is equivalent to

```
_foo:  
    ; some asm code
```

.if <exp>, **.else**, and **.endif** implement conditional assembly. If the <exp> is nonzero, then the assembly code up to the matching **.else** or matching **.endif** is processed. Otherwise, the assembly code from the matching **.else**, if it exists, to the matching **.endif** is processed. Conditionals may be nested up to 10 levels.

.include <string> opens the file named by the double quoted "string" and processes it.

<name> = <exp> assigns the value of the expression to the name.

HC11 Instructions

You should refer to the Motorola documentation for a comprehensive reference on the HC 11 Instructions. Instruction operands, if they exist, take the form of:

- an expression, which is usually a constant or memory address expressed using the above format and operators, or

- **bclr**/**brclr** and **bset**/**brset** use different syntax from that of the Motorola assembler:

```
bclr [opnd],#mask  
brclr [opnd],#mask,label
```

and

```
bset [opnd],#mask  
brset [opnd],#mask,label
```

[opnd] must either be a direct page reference (e.g., *_foo, or *0x10), or an indexed operand (e.g., 12,x or 3,y)

- an index addressing mode: a displacement off the X or Y register. For example:

```
4,x  
0,Y
```