ICC11

DOS based C Cross Compiler for MC68HC11

User Manual

version 0.43

Feb. 8th, 1993

LICENSE AGREEMENT

This is a pre-general release version 0.43 of the ImageCraft C Cross Compiler for the Motorola MC68HC11 processors. There is no charge for a license to use this release of the product. This product is protected by copyright laws and is not in the public domain. This license and the use of this product expires when the general release (version 1.0) becomes available to the general public. This license will not cover the general release version, nor any subsequent releases.

The general release will contain a relocatable assembler and linker, and is *estimated* to cost around $50.

WARRANTY INFORMATION

There is no warranty available on this version of the ImageCraft C Cross Compiler. This product and all related documentation are provided "AS IS." ImageCraft and the authors of this product do not warrant that this product will be bug-free, and since this is a pre-release version, there are likely to be bugs. UNDER NO CIRCUMSTANCES, INCLUDING NEGLIGENCE, SHALL IMAGECRAFT OR THE AUTHORS OF THIS PRODUCT BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OF OR INABILITY TO USE THIS PRODUCT.

If you do not agree with the above licensing agreement and warranty information, you have no right to use this product and you must destroy this copy of the product immediately.

Introduction

The ImageCraft C Cross Compiler for the MC68HC11 is a set of low cost standard conformance tools for programming in C and assembly on Motorola's 68HC11 series of microcontrollers. It consists of the following component programs:

1. icc11.exe
   - the driver program.
2. icpp.exe
   - the C preprocessor.
3. iccom11.exe
   - the compiler.
4. ias11.exe
   - the assembler.
5. crt.s
   - C runtime assembly file.
6. printf.c
   - simple printf-like function.

This manual does not describe how to write C or assembly programs. Any textbooks on ANSI standard C can be used for reference on C, and the assembler is based on the Motorola freeware version whose syntax is fairly well known and documentation for it

can be obtained in many places. These programs accept file paths in either the Unix-style forward slash format or the DOS-style backward slash format.

If you use this product, please send us a message with your contact address so we may send you announcements of availability of new versions. Copies of the product on a 3 1/2" IBM disk can be obtained for a "shipping and handling" charge of $9.95. Of course, even though this version is available for free, a small contribution of between $5 to $10 will give us encouragement and incentive to continue development of this product (please make checks payable to Christina Willrich).

Currently, the best way to register and contact us are:

```
e-mail: imagecft@netcom.com, or

ImageCraft

P.O.Box 64226, Sunnyvale, CA 94086-9991
```

Please report bugs to the contact address above. Machine configuration and a copy of the source code will help us to track down the bug. The compiler can generate information such as tables of variables' stack offsets, and line numbers as labels in the assembler output, both of which are helpful in tracking down your programming bugs and pinpointing compiler errors.

Since this release contains several unimplemented features, we would also appreciate it if you inform us which ones are most important to you, so that they can receive a high development priority.

Machine Requirements and Program limitations

This program runs in 8086 real modes under DOS, Windows VDM, OS2 VDM and probably most DOS emulators on the popular Un*x and Mac*ntosh machines. A 32 bit protected mode version that runs on OS2 2.X and DOS can be obtained from us. The 32 bit version supports HPFS long filenames, wildcard expansion, has access to large memory space and executes faster than the 16 bit version, but otherwise is identical to it. The compiler should also run inside Integrated Development Environments such as IDEAL and Brief.

Unlike most other DOS-based compilers that have separate front end and back end programs, this is an integrated compiler in that the front end and the back end are a single monolithic program. This improves execution speed at the expense of requiring more conventional memory. The compiler does not use extended or expanded memory, so if you encounter an out-of-memory problem, your recourse is either: a) freeing up more conventional memory, b) reducing the size of the program, or c) acquiring the 32 bit version of the compiler.

The language preprocessor is close to ANSI C conformant and the C language accepted by the language processor is ANSI C conformant. The code generator does not yet support floating point code generation. *long* data type is the same size as the *int* data type, which is 2 bytes. In addition, the lcc compiler front end defines *char* data type as equivalent to *signed char*, which is unfortunate for the HC11 target since all

computations smaller than *int* are promoted to *int*, and *signed char* promotions to *int* are more expensive than *unsigned char* promotions on the HC11.

There is no linker, librarian, debugger,or standard C headers and library in this release. Lack of a linker is not too much of a limitation since the assembler, compiler, and compiler driver have been written so that files of multiple types can be compiled and assembled together. A limited version of printf is provided.

Some of these limitations may be fixed in a later release.

The generated code should run on any version of the HC11, in both single-chip and expanded modes.

Installation

Put all the .exe files in a directory in your path. Crt.s should be placed either in the same directory as your source or in the directory pointed to by the environment variable ICC11_LIB. Header files should be placed in the directory pointed to by the environment variable ICC11_INCLUDE.

To verify the system is set up correctly, type "icc11 -o dhry dhry.c printf.c" on the command line. You should see some (harmless) warnings from the preprocessor and an output file dhry.s19 should be produced.

Program Operation

This is a traditional compiler system where you invoke the compiler driver with your C and assembly source files, and if there is no error, an output file is produced. The output is a Motorola hex format file, suitable to download to a monitor program such as BUFFALO.

File paths can use either forward slash or backward slash format. In addition to the current directory, the preprocessor searches for header files in a directory specified by the environment variable ICC11_INCLUDE if it exists. The compiler driver also directs the preprocessor to search for system header files in /lcc/include by default.

Even though there is no separate relocatable linker in this release, the assembler can take multiple assembly source files and thus can be used in lieu of one. The code generator has been written such that any local symbol defined in a source file is distinct from generated local symbols in other source files (basically, local names are appended with the source file name) so that there should not be any local symbol name conflict.

The driver automatically invokes the assembler to assemble the file *crt.s* before other files that you specify. *Crt.s* contains several routines that the code generator emits for code that is too large to be generated inline. For example, multiply, divide and structure assignments routines are placed in there.

Crt.s also contains the startup code. The starting address for the code section (usually start of free RAM, or EEPROM), the starting address for data (usually external RAM or HC11 internal RAM) and the initial stack address are specified in here. You should modify these addresses to match your environment. This code fragment defines the

starting address of the program, and the initial value of the stack pointer, then jumps to the C function *main*.

To use the *printf* function, you must define a function called *putchar* that outputs a single character and compile *printf.c* (or *printf.s*) with your files. The file *crt.s* contains a version of *putchar* that uses BUFFALO to do the output. You must replace it with something suitable for your system. This implementation of *printf* is very primitive. Please refer to the source code comment to see what it does support.

For systems with small amount of memory, an assembler listing should be generated to determine whether the program fits in the memory space or not. The compiler does not place any limit on the number or size of the local variables. However, due to the architecture of the HC11, you should consider placing large objects in static storage rather than on the stack since an indexed addressing mode can only access within 256 bytes of the frame pointer (as mentioned, the compiler does generate correct code to access objects with offset larger than 256 bytes off the frame pointer).

Debugging

The debug command line switch (-g) directs the compiler to emit labels of the form *filename.line* in the assembly output file where *line* is the source line number. The assembler listing file contains these labels and their actual code offset. Using these code offsets, breakpoints can be set in a debugger.

In the assembly output file, after each function label, the compiler emits a list of local variables and parameters with their corresponding offset off the frame pointer IX, as a comment block. Along with the debugging line labels above, it is relatively easy to inspect the source code and the listing file for debugging purposes.

Code Quality and Optimizations

Although the compiler does not perform register allocation or other resource-demanding global optimizations, it does perform basic block Common Subexpression Elimination in the front end, and branch shortening in the back end. Function arguments are passed in preallocated areas on the stack so no stack adjustment code is necessary after function calls. In addition, the compiler has a built-in peephole optimizer that walks through the generated code and eliminates redundant loads and stores, and replaces other inefficient code fragments with something shorter and faster.

Space or time critical code can be either be written as C callable assembly routines, or assembler instructions can be embedded directly in the C source files.

Interfacing C With Assembly Routines

C global function and data names are prefixed with an '_', and arguments are pushed right to left and converted to their "natural size." That is, char and short are converted to int, and float is (or will be) converted into double. All arguments are passed on stack. D register contains the return value, if any. IX and IY registers can be used freely in an assembly routine. Argument stack space is allocated on function entry and thus does not need to be freed by the callee.

The assembler supports the "sect" pseudo op. "Sect 0" is the code section and "sect 1" is the data section. Each section has its own program counter. This is especially useful for using the HC11 in single chip mode where you may assign the code section to internal EEPROM and the data and stack sections to the internal RAM.

The compiler supports embedded assembly in the C source code. The format is

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

"string" may contain variable references in the form of "%name." If name is a variable that is defined, the generated assembly replaces %name with the assembler references. If name is not a visible variable, then the literal "name" will be output. C escape sequences such as '\n' may be used inside the string. Note that an embedded assembly instruction must begin with a space or tab if it does not start with a label. Otherwise the assembler will generate an error. For example, the following fragment:

```
int external;
foo(int param)
        {
        int local;

        asm(" ldd %param");
        asm("label: addd %local");
        asm(" std %external");
        asm(" ldd %xxx");           /* xxx is not a variable */
        ...
```

may generate something like:

```
        ldd     2+2,x
label:  addd    2-2,x
        std     _external
        ldd     xxx
```

You should not use "%?" inside the asm string, the output is meaningless. Note that the peephole optimizer does not optimize embedded asm code.

Command Syntax

icc11 [options] *filenames*

compiles the input files by invoking the appropriate processors to handle the input file types. The output is a Motorola hex format file, with an extension of .s19. Acceptable file types are: .c for C source file, .i for C source file after preprocessing, and .s file for assembly source file. Options are:

- -v verbose, prints out each action before execution. If specified more than once, no action gets executed.
- -o file names the output file, default iccout.s19
- -l creates an assembly listing file with name <output file name>.lst
- -E preprocesses a .c file into an .i file
- -S compiles into assembly but does not assemble.

- -s silence mode. Will not print out each file name as the files are processed.

The following flags are passed directly to iccom11; please refer to their descriptions under that section.

- -P
- -A
- -w
- -g
- -O

icc11 also accepts other options, their behaviors are currently undefined.

icpp [options] *input [output]*

A complete description of this preprocessor and the flags can be found in cpp.mem. Some of the more important flags are:

- -D*name[=value]*
  define the word *name* to have the *value*. If *value* is not specified, then 1 is assumed
- -I*directory*
  add *directory* to the search path for locating include files.

By default, **icc11** specifies the *-I/icc11/include* to **icpp**.

iccom11 [options] *input [output]*

Compiles the preprocessed C source into assembly file. Options are:

- -v prints out compiler version string.
- -O0 disables the peephole optimizer.
- -P generates function prototypes for functions encountered. Useful for converting non-ANSI programs to ANSI C.
- -A checks for strict ANSI conformance. If specified more than once, emits additional warnings.
- -w do not generate warning.
- -g generates debugging line labels.

ias11 [options] *filenames*

Assembles the input files into a hex format output. All modules of a program must be specified, otherwise, an undefined symbol value error will result. Options are:

- -l generates a listing file.
- -o *file* names the output file. Default is iccout.s19.
- -s creates symbol table dump.
- -c includes cycle counts in listing file.
- -cre includes cross reference in listing file.

Known Bugs

- _asm() only works inside a function definition. It also ought to work outside a function.
- The debugging line number labels only use the base file name, and thus executable code defined inside an include file will not use the correct file name in the labels.

REXIS

A prime reason this compiler is written is so that hobbyists can experiment with **REXIS** (tm) (**R**eal time **EX**ecutive for **I**ntelligent **S**ystems). This release does not yet fully support REXIS since it is lacking several library functions. Here is an excerpt from the REXIS user manual:

> **REXIS** (**R**eal-time **EX**ecutive for **I**ntelligent **S**ystems) is a small multitasking preemptive executive for embedded microcontroller systems. Based on both the designs of traditional real-time kernels and subsumption architecture as developed at the MIT Artificial Intelligence Laboratory for programming autonomous robots, **REXIS** provides a framework for implementing control programs for intelligent systems such as those used in robotic and distributed networks. It includes functions for managing tasks, messages, events, semaphores, scheduling, and memory allocation.
>
> **REXIS** is written in portable ANSI C with a small assembly module for fast and efficient task switching. It is currently targeted for Motorola MC68HC11 microcontroller systems with at least 24K of RAM. Other targets are under consideration.

Please contact us if you are interested in **REXIS** or in using this compiler system with **REXIS**.

Acknowledgments

Most of the components in this current version are either in the public domain, or are copyrighted materials with permission for distribution. **icc11** and the front end component of **iccom11** are based on the lcc distribution written and copyright by David Hanson. The copyright information on these components are stated in the file *readme.lcc*. **icpp** is based on the public domain DECUS C preprocessor written by Martin Minow. The file *readme.cpp* is the readme file from the DECUS distribution. **ias11** is based on the public domain assembler from Motorola. If you want information on how to obtain the sources for unmodified versions of these programs, please look in the relevant readme files or contact us.

The c ode generator component of the compiler, **iccom11**, is entirely of Richard Man's creation and is not based on any existing compiler technology. Christina Willrich is the resident English language / techwriting guru and helped to prepare anything that Richard has ever published or distributed. Thanks to Ariane (age 3) for letting us finally use the computer after she played "*Busytown*" for the zillionth time.