APPLICATION NOTE:
COSMIC V4.1x 68HC12 C COMPILER PACKAGE

This application note is intended to aid understanding of how to get the best results from an evaluation of the COSMIC 68HC12 V4.1x C compiler package.

Evaluation Criteria

In general, most evaluations follow the same form and have similar criteria and below we list the criteria which are most often specified as being of primary importance to end-users.

Reliability

1. The C compiler package must give **reliable operation**. The COSMIC 68HC12 compiler has been in the marketplace for three years and is field-tested at hundreds of sites worldwide. Reliability is a key feature of the COSMIC compiler. However, there are an infinite number of test cases needed to exhaustively remove all possible defects; if you find a defect during your evaluation, please report it via email to: ***support@cosmic-us.com*** or call (+) **781 932-2556** and ask for technical support.

2. **It must build your application code**; we cannot guarantee it will immediately compile your C code or assemble your assembly language code without errors. If you are migrating C code from a different microcontroller and your code compiled with a COSMIC V4 compiler, it should be a relatively simple job to recompile using the COSMIC V4 68HC12 compiler. If, however, you have been using another vendor's C compiler, do not expect to be able to recompile without changes. Most embedded compilers have added language "extensions" to support features not supported in ANSI C that are important to embedded systems programmers. The table below lists the current extensions supported in the COSMIC 68HC12 C compiler:

Language Extensions

Extension Syntax	Description	6812
_asm()	in-line assembler function. Can be used in C expressions In-line code must be inside a C function.	✔
@<address>	specify code/data at absolute address, *address*	✔
@bool	function returning with condition codes already set	✔
@builtin	in-lined or intrinsic function (not supported in V4.1C)	✔
@dir	force direct page addressing for static data. This can be forced as a compile-time option using the **+zpage** option at compile-time	✔
@eeprom	special write sequence (if EEPROM available)	✔
@far	32-bit addresses (or bank switching)	✔
@fast	Fast function calling. This can be forced as a compile-time option using the **+fast** option at compile-time	✔
@interrupt	C interrupt function	✔
@near	16-bit addresses (default except for 68HC05)	✔
@nostack	function using static model instead of stack. This can be forced as a compile-time option using the **+nostk +st?** options at compile-time	X
@pack	pack structures or locals when even alignment (6816)	X
@port	I/O (not memory-mapped, or need special mode)	✔
@regsafe	don't save/restore registers on function entry/exit	X
short float	Fixed-point DSP support (6816 only)	X
@tiny	8-bit addressing (zero page, default for 68HC05)	X

Language Pragmas

Pragma	Description	6812
#pragma asm **#pragma endasm**	In-line assembly language. Assembler code can be inside or outside a C function.	✔
#asm **#endasm**	alias for #pragma asm	✔
#pragma section <name>	Place code/data in new program section *name*	✔
#pragma debug [on/off]	specify debugging to be on or off	✘
#pragma space <sc>	set default space modifiers	✔

✔ -- feature is supported ✘ – feature is not supported

If you are going to make use of language extensions, we recommend you define the extensions as a series of macros in a separate source file; this will help you keep your application code portable and compiler independent. For example:

```
#ifdef COSMIC
#define INTERRUPT @interrupt
#endif
..
INTERRUPT void isr();
      {
      ...
}
```

Note that by using the *@keyword* syntax to represent special things to the compiler, you are still free to use *keyword* as an identifier in your source code. Also, use #pragma very carefully! Most C compilers will, if they don't recognize the #pragma syntax, simply ignore it and if your code relies on the pragma being recognized, you can get a successful compile but your application won't run.

The COSMIC 68HC12 assembler, ***ca6812***, conforms to the Motorola MCUasm standard assembly language format, the current Motorola standard. Unfortunately, there has typically been no defined industry standard for assembly language format, so if you are migrating assembly language code into COSMIC format, you may have to invest some time and effort translating your code.

Most of the conversion effort will involve pseudo-ops, things like storage directives and macro formats. The actual assembler instructions will most likely need no conversion. If you make extensive use of (complex) relocatable arithmetic in your assembly source code, you may have to re-code this part of your application.

Tools to help you convert from your old assembler format to COSMIC V4 format are: PERL, sed or awk. Most of these utilities are available free on the Internet.

Compiler Performance

3. **It must give good performance**. Some compiler vendors will tell you they have lots of different optimization levels available, that you can control; while this may be true, the net effect of these levels on a limited architecture, like the 68HC12, may be minimal or, even worse, detrimental to correct operation! COSMIC's approach to optimization is to let the C compiler do most of the work and give the user control over very few options, to reduce complexity and improve reliability.

Performance Oriented Compile-Time Options

The compile-time options that will make a difference to application performance are:

-no do not use the optimizer. If you want to compile code with this option, be aware that your code will not be optimized, resulting in larger size and slower execution. By default, the optimizer is enabled. If you are wary of optimizers, because of previous experiences with highly optimizing compilers, this flag may be a good starting point for you. We recommend you do not use this option, unless you are certain the optimizer is the source of problems.

+nowiden this flag disables the widening of *char* values to *int* values and *float* to *double* across function calls. By default the compiler will widen, to adhere to ANSI C. If your code makes heavy use of *char*-sized or *float*-sized function arguments, compile with this flag. It is safe to compile with this option all of the time.

+fast In-line machine library calls for long integer arithmetic and integer switches. This flag can cause your code to execute faster at the expense of larger code size. If application execution speed is more important than application size, try compiling with this option.

+sprec if you need floating point support, but can make do with *float* precision (32-bits total), this flag will force all floating point arithmetic to be done in strict single-precision. By default, the compiler will promote *float* (32-bit) to *double* (64-bit) in expressions where *floats* and *doubles* are mixed and such arithmetic is carried out in double precision, which is typically 3 to 4 times slower than single precision.

+zpage this flag causes all static data, declared outside a C function body, to be allocated storage in the direct or zero page (usually the first 256 bytes of address space), where it can be addressed using direct addressing. Direct address access is one byte shorter than addressing data outside direct page memory. Note the 68HC12 I/O register block usually occupies all of direct page memory, so you will not be able to use this flag unless you have mapped the I/O block to a different 2kb boundary other than 0x00.

If you can make use of direct page memory for application data, but you aren't sure how big your static data is, you can compile with the +zpage option and at link time you tell the linker to check for zero page overflow by specifying the maximum size of the zero page program section, which is named the .bsct section. In you linker command file you need to have the following line:

```
+seg .bsct -m256
```

which will cause the linker to generate an error message when more than 256 bytes of data are being allocated into the .bsct section.

The advantage of using the +zpage compile-time flag, is you don't need to modify your source code to get the desired effect. If, however, you want fine control over which data is placed in direct page memory or if you get an overflow error message from the linker, you should use the @dir type qualifier attached to the data declaration to selectively place commonly used data into direct page memory:

For example,

```
#define FASTMEM @dir
```

```
unsigned char count = 1;     /* count is in the regular .data section */
FASTMEM unsigned char fastcount = 1; /* fastcount is in .bsct section */
```

For a description of default program section names used by the compiler/linker see Program Section Names description below.

Performance Summary

In summary, you should evaluate your use of the following compile-time options:

```
> cx6812 +nowiden test.c          /* use +nowiden all the time */
> cx6812 +nowiden +sprec test.c   /* use +sprec if using floating point, and single-precision
                                     (32-bit) floating point is adequate */
> cx6812 +nowiden +fast test.c    /* use +fast if speed is more important than size */
> cx6812 +nowiden +zpage test.c   /* use +zpage if you need to speed up access to most
                                     frequently used static data by placing it in direct page
*/
```

Ease Of Use

4. **Ease of Use.** How easy is it to use the compiler package? A question often asked is "does the compiler run under Windows?" The answer is "yes!" but compilers are not graphical user interfaces; most users decide what compile-time options are required and will use a Make utility and Make file to build the application, so the compiler just becomes a "black-box" that either compiles application code without error or it generates an error file if there are errors.

There are two versions of the C compiler package available; one version runs on Windows 3.1/3.11/95/NT systems and under DOS and runs as a 16-bit application; the other version runs as a 32-bit application and provides support for long filenames, so it will only run under Windows 95 or Window NT.

If you want to run a Make utility or the C compiler directly, without leaving Windows, what you really need is a good Windows-based code/project editor. There are a couple of options:

(a) Premia Corporation's Codewright Professional editor is an excellent code/project editor and I.D.E. that supports programming in Assembly, ADA, C, C++, Cobol, Java, HTML, Pascal, Perl, and Visual Basic, so it serves your wider programming needs. It also offers vi, Brief or CUA emulation modes, so if you are used to using MultiEdit, you will find Codewright easy to use.

COSMIC has integrated all it's C compilers with Codewright, so that any compile/assemble/link-time error messages are automatically detected and displayed; a simple mouse-click on the error line will put you right at the offending line in your original source code. You can compile/assemble and link your code without leaving Windows, or you can invoke a Make to build your application.

You can get a full, time-limited Codewright demonstration at Premia's web-site: *http://www.premia.com.* Or call them at 1 888 477-3642 or (+) 503 641-6001.

If you decide to use Codewright, you can buy it directly from COSMIC.

(b) The GNU Emacs editor is now also available for Windows 95/NT. Emacs is widely used on UNIX workstations and is available free from the Free Software Foundation. You can get pre-built binaries and complete source code for Emacs from a number of Internet sites (check your local university web or ftp sites). Take a look at *http://www.cs.washington.edu/homes/voelker/ntemacs.html* where you will find a lot of information about Emacs. A couple of COSMIC customers have implemented Emacs error parsers, so that compile-time errors are automatically detected by Emacs. The lack of commercial grade technical support service for Emacs is it's biggest drawback.

If you need a high quality Make utility, we recommend you look at Opus Software Make. You can download a demo from *http://www.opussoftware.com* or call them on 1 800 248-6787. You can also use Microsoft Make (Nmake) or virtually any other Make you are used to using.

Checking Application Code Sizes

You are at the point of having your application built and running OK. Now you want to collect some data on program sizes. Here you have to be sure you are looking at real program section sizes. To look at the sizes of object files (.o files) or executable files (.h12 files), use the **cobj** utility program included with the compiler. This utility is an object file inspector and can print out lots of useful information about your object files, including the constituent program section sizes.

The option that is of most interest in determining program section sizes is the -n option. We suggest you look at object file (.o) section sizes first as this gives you a good indication of compiler code efficiency. Then you need to look at the executable file (.h12) section sizes as this is the final linked executable that will be going into your product. Remember that the combined sizes of the .text, .const and .init sections must fit into your ROM/EPROM and the combined sizes of the .data, .bsct and .bss sections must fit into available RAM and you must have space for a stack.

To print out the sizes of the program sections defined in the relocatable object file prog.o:

```
> cobj -n prog.o

sections:

.text: hilo
      111 data  bytes at 108
       48 reloc bytes at 219
.bss: bss hilo
      516 reserved bytes
```

The above tells you there are 111 bytes in the .text section and 516 bytes in the .bss section of the object file. There are no other program sections defined in this example.

Now if the object file is linked with the start-up file, crts.o, and with run-time libraries, we can create the fully linked executable file, prog.h12 using the linker command file prog.lkf:

```
> clnk -o prog.h12 prog.lkf
> cobj -n prog.h12

sections:

.text: hilo code, at address 0xf000
      133 data  bytes at 284
```

```
.const: no attribute, at address 0xf085
        0 data  bytes at 417
.data: no attribute (init), at address 0x0
        0 data  bytes at 417
.bss: bss hilo, at address 0x0
      516 reserved bytes
```

The above data tells you the .text section has grown to 133 bytes and the .bss section is still at 516 bytes. The .text section is bigger because *prog.o* was linked with the C runtime startup file, *crts.o*, which contains exactly 22 bytes of code. There were no unresolved references to library routines in this example.

A common mistake is to look at the sizes of the .o, .h12 or .hex files at the DOS or Windows directory level. These sizes include additional information, such as debugging and relocation information and are therefore misleading. The only meaningful, and easiest to obtain, results are those generated by the **cobj** utility.

Checking Application Code Speed

To perform a full real-time analysis of target code execution speed, you probably need to run your application on real target hardware or on an in-circuit emulator. If however, you need to perform a quick timing analysis and you have an evaluation copy of COSMIC's ZAP/SIM12 simulator/debugger, you can use the simulator to get timing information. When single-stepping or when running to a breakpoint, ZAP/SIM12 will provide a cycle count in the Register window which can be converted to a time figure based on the target system CPU bus speed. If your application is not entirely I/O driven, you can stub out code that relies on hardware being present, and run the whole application under the simulator; the ZAP/SIM12 Analyze/Performance menu system gives a histogram of cycle counts for the whole application on a function or source file basis.

The remainder of this application note supplements the current COSMIC compiler user documentation.

Program Section Names

The following program section names are defined by the compiler as defaults:

Default Program Section Names	Description
.text	Function code which normally resides in ROM/EPROM
.const	Literal data: strings, constants, switch tables, C data declared as *const*. This default can be overridden using the -nocst flag at compile time to force all constant data into the .text section. This program section is normally located right after the .text section
.data	Initialized static C data which is normally resident in RAM, but whose initial values are normally held in ROM; this includes static data that is initialized to zero. This program section is normally located in RAM, but not in direct page RAM
.bss	Un-initialized static C data which is normally resident in RAM. The .bss section (block started by symbol) is really just a 32-bit counter that counts the number of bytes of un-initialized data, and at start-up time, the C run-time start-up files, *crts.s* or *crtsi.s*, initializes the .bss section to zeros. .bss data can be forced into the .data section at compile-time using the +nobss option.
.bsct	Initialized or un-initialized static C data declared using @dir type qualifier or static data contained in source files compiled with +zpage flag. This program section is normally located in direct page RAM (first 256 bytes of the address space)
.eeprom	Reserved for data declared using @eeprom
.init	Copy of initialized data sections (.data & .bsct sections) to be located in ROM and copied to RAM at start-up using *crtsi.o* ; by default .init is appended after the first .text section (see description below entitled "Initialization of Static Data"). This section is created by the linker when you link with *crtsi.o* or reference the symbol **__idesc__** in your code
.debug	Debugging section. Used by the ZAP C source-level debugger or other third party debuggers. Not of interest to most users.

User Defined Program Sections

It is possible to redirect the default program sections to any user defined section, at the C level, using the following pragma syntax, which is only supported in the V4.1x compiler releases:

#pragma section <*attribute*> <*qualified_name*>

where <*attribute*> is either empty or one of the following sequences:

const
@dir

and <*qualified_name*> is a section *name* enclosed as follows:

(*name*) -- parenthesis indicate a code section

[*name*] -- square brackets indicate un-initialized data
{*name*} -- curly braces indicate initialized data

A section *name* is a plain C identifier which does not begin with a period (.) and which is no longer than thirteen characters. The compiler will automatically prefix the section name with a period when passing the section name to the assembler. To switch back to the default section names, just omit the section name in the <*qualified_name*> sequence:

```
#pragma section ()              /* switch back to default .text section */
```

Examples:

```
#pragma section (.code)         /* executable instructions redirected to section named .code */
#pragma section const {string}  /* constant strings and data redirected to section named .string */
#pragma section [.udata]        /* un-initialized static data redirected to section named .udata */
#pragma section {.idata}                /* initialized static data redirected to section named .idata */
#pragma section @dir [.zpage]   /* initialized & uninitialized direct page data redirected to section .zpage */
```

Note that {*name*} and [*name*] are equivalent for the const section as it is always considered initialized.

A practical C example involves defining a program section for interrupt vectors, which have to be placed at a specific location in target system memory. The C source file, *test-vec.c*:

```
#include <stdlib.h>

extern void _stext();         /* startup routine */
extern void sci_rcv();        /* SCI character receive handler */
extern void spi_rcv();        /* SPI receive handler */

#pragma section const {vectors}            /* define a new section for constants
                                              called vectors */

void (* const _vectab[])() = {                 /* 0xFFCE */
      0,          /* Key Wake Up H   */
      0,          /* Key Wake Up J   */
      0,          /* ATD             */
      0,          /* SCI 2           */
      sci_rcv,          /* SCI 1     */
      spi_rcv,          /* SPI       */
      0,          /* Pulse acc input */
      0,          /* Pulse acc overf */
      0,          /* Timer overf     */
      0,          /* Timer channel 7 */
      0,          /* Timer channel 6 */
      0,          /* Timer channel 5 */
      0,          /* Timer channel 4 */
      0,          /* Timer channel 3 */
      0,          /* Timer channel 2 */
      0,          /* Timer channel 1 */
      0,          /* Timer channel 0 */
      0,          /* Real time       */
      0,          /* IRQ             */
      0,          /* XIRQ            */
      0,          /* SWI             */
      0,          /* illegal         */
      0,          /* cop fail        */
      0,          /* cop clock fail  */
      _stext,     /* RESET           */
      };
```

```
#pragma section const {}            /* now switch back to the default section
                                       name for constants, .const */

const int a = 255;                  /* variable 'a' is in .const section */

void main() /* do nothing example */
    {
    int b;

    b = a;
}
```

When compiled, *test-vec.c* generates the following assembly language listing file (test-vec.ls) and object file *test-vec.o*; note that the vector table, vectab, is defined in its own program section, named .vectors, which can be located to the proper location in target memory using the linker.

```
 1                  ; C Compiler for MC68HC12 [COSMIC Software]
 2                  ; Version V4.1f - 03 Nov 1997
 3                  .vectors:       section
 4  0000               __vectab:
 5  0000 0000              dc.w    0
 6  0002 0000              dc.w    0
 7  0004 0000              dc.w    0
 8  0006 0000              dc.w    0
 9  0008 0000              dc.w    _sci_rcv
10  000a 0000              dc.w    _spi_rcv
11  000c 0000              dc.w    0
12  000e 0000              dc.w    0
13  0010 0000              dc.w    0
14  0012 0000              dc.w    0
15  0014 0000              dc.w    0
16  0016 0000              dc.w    0
17  0018 0000              dc.w    0
18  001a 0000              dc.w    0
19  001c 0000              dc.w    0
20  001e 0000              dc.w    0
21  0020 0000              dc.w    0
22  0022 0000              dc.w    0
23  0024 0000              dc.w    0
24  0026 0000              dc.w    0
25  0028 0000              dc.w    0
26  002a 0000              dc.w    0
27  002c 0000              dc.w    0
28  002e 0000              dc.w    0
29  0030 0000              dc.w    __stext
30                  .const: section
31  0000               _a:
32  0000 00ff              dc.w    255
33                  ; 1 #include <stdlib.h>
33                  ; 2
33                  ; 3 extern void _stext();               /* startup routine */
33                  ; 4 extern void sci_rcv();              /* SCI character receive handler */
33                  ; 5 extern void spi_rcv();              /* SPI receive handler */
33                  ; 6
33                  ; 7 #pragma section const {vectors}              /* define a new section for constants,
33                  ; 8     called vectors */
```

```
33                  ; 9
33                  ; 10 void (* const _vectab[])() = {                          /* 0xFFCE */
33                  ; 11    0,                  /* Key Wake Up H   */
33                  ; 12    0,                  /* Key Wake Up J   */
33                  ; 13    0,                  /* ATD             */
33                  ; 14    0,                  /* SCI 2           */
33                  ; 15    sci_rcv,            /* SCI 1           */
33                  ; 16    spi_rcv,            /* SPI             */
33                  ; 17    0,                  /* Pulse acc input */
33                  ; 18    0,                  /* Pulse acc overf */
33                  ; 19    0,                  /* Timer overf     */
33                  ; 20    0,                  /* Timer channel 7 */
33                  ; 21    0,                  /* Timer channel 6 */
33                  ; 22    0,                  /* Timer channel 5 */
33                  ; 23    0,                  /* Timer channel 4 */
33                  ; 24    0,                  /* Timer channel 3 */
33                  ; 25    0,                  /* Timer channel 2 */
33                  ; 26    0,                  /* Timer channel 1 */
33                  ; 27    0,                  /* Timer channel 0 */
33                  ; 28    0,                  /* Real time       */
33                  ; 29    0,                  /* IRQ             */
33                  ; 30    0,                  /* XIRQ            */
33                  ; 31    0,                  /* SWI             */
33                  ; 32    0,                  /* illegal         */
33                  ; 33    0,                  /* cop fail        */
33                  ; 34    0,                  /* cop clock fail  */
33                  ; 35    _stext,             /* RESET           */
33                  ; 36    };
33                  ; 37
33                  ; 38 #pragma section const {}              /* now switch back to the default section
33                  ; 39                                                name for constants, .const */
33                  ; 40
33                  ; 41 const int a = 255;                    /* variable 'a' is in .const section */
33                  ; 42
33                  ; 43 void main()          /* do nothing example */
33                  ; 44    {
34                          switch  .text
35  0000              _main:
36      00000002    OFST:   set     2
38                  ; 45    int b;
38                  ; 46
38                  ; 47    b = a;
39  0000 fc0000             ldd     _a
40                  ; 48 }
41  0003 3d                 rts
42                          xdef    _main
43                          xdef    _a
44                          xdef    __vectab
45                          xref    _spi_rcv
46                          xref    _sci_rcv
47                          xref    __stext
48                          end
```

To locate the .vectors program section at link time, include the following line in your link command file:

```
+seg .text -b 0xf000 -n .text      # code section, .text, starts at 0xf000
+seg .const -a .text               # constants section, .const, follows .text section
+seg .data -b 0                    # initialized data section, .data, located at 0
+seg  .vectors -b 0xffce           # interrupt vectors section, .vectors, located at 0xffce
test-vec.o
```

Initialization of Static Data

Static data (data defined outside a function) which is also given an initial value is handled separately from static data which is un-initialized:

Examples

```
      int i = 3;         /* i is initialized to 3 */
      int j = 0;         /* j is initialized to 0 */
      int k;             /* k is un-initialized */
      char obuff[4] = {'a','b','c','d'};  /* obuff is a 4-byte initialized character
array */
      char ibuff[4];     /* ibuff is a 4-byte un-initialized character array */
```

Why do you need to be aware of how the compiler/linker handles static data? The main reason is that a real embedded system has to be initialized somehow after a power on reset, for example, and important initial values have to be stored somewhere in non-volatile storage. Some embedded systems can initialize themselves by downloading the application across some sort of link and run out of RAM, but most embedded applications run out of ROM or EPROM. In a ROM-based system there must be RAM available to hold data values that can be written to, and to hold the stack. If your application code explicitly initializes static data, as with data items i, j and obuff above, then when the embedded system is initialized the initialized static data must be set to the initialized values otherwise the application will not run correctly. How is this done? The system has to get these initialized values from somewhere, and the obvious place is to keep a copy of them in ROM or EPROM and to copy them into RAM upon initialization of the system.

The compiler puts initialized static data, which is not in the direct page, into the .data section – all the data defined in this section is initialized to the initial values you specified in your code. Static data which is defined using @dir or compiled using +zpage is placed in the .bsct section, and unlike .data, contains both initialized and un-initialized data.

If your application defines initialized static data, you have to reference the symbol **__idesc__** somewhere in your application code, or you need to link in the modified C run-time start-up file *crtsi.o* instead of the default start-up file *crts.o*. This causes the linker to create a program section, named .init, into which it makes a copy of the .data and .bsct sections and any other used-defined segments (see -id linker option) which contain initialized static data. The linker locates a .text section, which by default is the first defined .text section but which can be over-ridden using the -it option to the linker, to act as a host for the .init section. After it has scanned the whole executable file to identify all initialized data sections which it has to copy, it also builds a descriptor containing the starting address and length of each such section; this descriptor is placed at the end of the host .text section and the copied sections are placed after the descriptor.

The process is completed by the *crtsi.o* code which copies the initialized static data from ROM into RAM.

So what? Well, what the above means is that if your application defines 2K of initialized static data, a copy of this 2K area has to be held in your ROM or EPROM, so you need to make sure your ROM is large

enough to hold program code, constants and initialized data. If you are short on ROM space, you may need to minimize your use of initialized static data.

Battery Backed RAM

In a ROM-based system, battery backup is commonly used to preserve certain areas of RAM. In this situation the embedded system has to differentiate between a “cold boot” and a “warm boot” when it starts up; on a cold boot, all data segments are initialized including battery backed RAM; on a warm boot, battery backed RAM is not initialized, but all other data segments are initialized. How is this accomplished using the COSMIC tools?

The *crtsi.s* startup code file allows you to *always* or *never* initialize a data segment at program startup and so we have to establish conditions that allow *crtsi.s* to recognize the two boot situations. There are a number of possibilities, but a simple one is to use a byte inside battery backed RAM, say the first byte, which contains a magic number; the magic number will be used to determine if the system should perform a cold or warm boot.

Once this is defined, *crtsi.s* has to be modified to test the magic byte before doing the initialization loop. Unfortunately, you may also have more than one program segment to initialize, so the problem is: how do you find which segment is the battery backed segment? The linker will help you here; when a segment is selected for initialization, either automatically by its section name (e.g. .data) or explicitly (with the -id linker option), it is entered in the descriptor created by the linker along with a flag byte, whose value is the ASCII code of the first *significant* letter of the segment name (either the section name, or the segment name given by the -n linker option, if this was used). The first significant letter means that any ‘.’ prefix character is ignored. Because the letter is encoded in the descriptor, the *crtsi.s* file can now test it against the expected character and thus locate the battery-backed segment. Here is a segment definition which can be used for battery backed data:

```
+seg .data -b 0x2000 -n .backed
```

Here the -id option is not necessary as the .data section name is directly recognized by the linker as a predefined initialized section name. The above line will encode the letter ‘b’ as the flag byte in the descriptor for the battery backed segment.

In the standard *crtsi.s* file, the flag is only tested against zero, to detect the end of the descriptor, using the following algorithm:

```
while (flag != 0)
      {
      copy_segment;
      skip to next segment;
      }
```

To handle battery backed RAM we need to modify this algorithm as follows:

```
while (flag != 0)
      {
      if (flag != ‘b’ || first_byte != magic)
            copy_segment;
      skip to next segment;
      }
```

So on a system boot, the battery backed area of RAM will NOT be initialized if its first byte is equal to the magic value. Note that it is easy to find the location of this first byte as this address is encoded in the descriptor which contains the destination address for the segment.

Below is a listing for a modified *crtsi.s*, for 68HC12 target, with the code added to handle a battery backed data segment:

```
;     C STARTUP FOR MC68HC12
;     WITH AUTOMATIC DATA INITIALISATION
;     Copyright (c) 1996 by COSMIC Software
;
      xdef  _exit, __stext
      xref  _main, __memory, __idesc__, __stack
;
      switch      .bss
__sbss:
;
      switch      .text
__stext:
      lds   #__stack    ; initialize stack pointer
      ldx   #__idesc__  ; descriptor address
      ldy   2,x+        ; start address of prom data
ibcl:
      ldaa  5,x+        ; test flag byte
      beq   zbss        ; no more segment
      bpl   nopg        ; page indicator
      leax  2,x         ; skip it
; start of extra code to support battery backed RAM
nopg:
      cmpa    #'b'      ; compare flag to 'b' code
      bne     cok       ; not equal, do copy
      ldab    [-4,x]    ; load first byte of ram
      cmpb    #MAGIC    ; compare with expected value
      beq     ibcl      ; found, do not copy and skip to next
cok:
; end of extra code to support battery backed RAM
      pshx              ; save pointer
      tfr   y,d         ; start address
      subd  -2,x        ; minus end address
      ldx   -4,x        ; destination address
dbcl:
      movb  1,y+,1,x+   ; copy from prom to ram
      ibne  d,dbcl      ; count up and loop
      pulx              ; reload pointer to desc
      bra   ibcl        ; and loop
zbss:
      ldx   #__sbss     ; start of bss
      clrb              ; complete zero
      bra   loop        ; start loop
zbcl:
      std   2,x+        ; clear byte
loop:
      cpx   #__memory   ; end of bss
      blo   zbcl        ; no, continue
      jsr   _main       ; execute main
_exit:
      bra   _exit       ; stay here
;
      end
```

Other Useful Compiler Features

Version and Flag Options:

If you want to find out what flag options a particular compiler package utility supports, just run the utility by itself without any options specified. For example, to list the C command driver options:

```
> cx6812

COSMIC Software Compiler Driver V4.1f
usage: cx6812 [options] files
    -a*>          assembler options
    -ce*          path for error files
    -cl*          path for listing files
    -co*          path for object files
    -d*>          define symbol
    -ex           prefix executables
    -e            create error file
    -f*           configuration file
    -g*>          code generator options
    -i*>          path for include
    -l            create listing
    -no           do not use optimizer
    -o*>          optimizer options
    -p*>          parser options
    -sp           create only preprocessor file
    -s            create only assembler file
    -t*           path for temporary files
    -v            verbose
    -x            do not execute
    +*>           select compiler option
```

The -vers option generates the version number of the executable program. To list the version and date of the compiler driver:

```
> cx6812 -vers
```

To get a version number list for all the executables included in the compiler package:

```
> version

-----------------
68HC12 C Compiler
Version: 4.1f
Date: 04 Nov 1997
-----------------
COSMIC Software Compiler Driver V4.1f - 03 Nov 1997
COSMIC Software C Cross Compiler V4.1g - 23 Oct 1997
COSMIC Software M68HC12 Code Generator V4.1f - 03 Nov 1997
COSMIC Software 68HC12 Optimizer V4.1e - 03 Nov 1997
COSMIC Software Macro-Assembler V4.1f - 03 Nov 1997
COSMIC Software Linker V4.1d - 03 Nov 1997
COSMIC Software Hexa Translator V4.1c - 23 Oct 1997
COSMIC Software Absolute Listing V4.1c - 23 Oct 1997
COSMIC Software Librarian V4.1c - 23 Oct 1997
COSMIC Software Absolute C Listing V4.1c - 23 Oct 1997
COSMIC Software Object Inspector V4.1c - 23 Oct 1997
COSMIC Software Print Debug Info V4.1c - 23 Oct 1997
COSMIC Software Bank Packing V4.1c - 23 Oct 1997
```

This last feature is designed to make it easier for COSMIC's support group to provide more efficient end-user technical support, by allowing for quick verification that you are using versions of the utilities that have been qualified as a full release. If your version numbers do not match up to a full qualified release, it may be a source of problems.

Listing Files

By default, the interspersed C/ASM listing file generated by the compiler includes only C source lines that cause executable assembler code to be generated; these are short-form relocatable listings.

```
> cx6812 -l test.c      /* generate short-form listing in test.ls */
```

If you want a full listing with all your original C source lines included, compile as:

```
> cx6812 -l -gf test.c  /* generate full interspersed C/ASM listing */
```

If you want to see an absolute interspersed listing rather than a relocatable listing, you need to compile, assemble and link your test code, to produce an executable file *test.h12*; when you compile make sure you specify the "-l" option to generate the relocatable listing file, *test.ls*. To generate the absolute listing file, just run the **clabs** utility program:

```
> clabs test.h12
```

The absolute listing is now in a file called *test.abs*.

You can specify an absolute pathname for output listing files using the "-cl *" flag at compile-time.

```
> cx6812 -l -cl \c\6812\tests test.c
```

causes the listing file *test.ls* to be placed in the directory \c\6812\tests, which must exist.

Generating Efficient Bit-addressing Instructions

The compiler makes extensive use of 68HC12 bit instructions (bclr, bita, bitb,bset, brclr, brset), where possible. Bit instructions have been enhanced on the 68HC12, compared to the 68HC11, and work well in direct page, indexed and extended addressing modes. For most 68HC12 family devices, the memory-mapped I/O register block is mapped at address 0x0000 and the C compiler will use direct page addressing for accessing bits in the control registers. If you map the register block outside of direct page memory (first 256 bytes of the address space), the compiler will use extended addressing to access bits; extended addressing is one byte longer than direct page or indexed addressing.

If you are using the 68HC812A4 you should include the header file *ioa4.h* or the header file *iob32.h* if you are using the 68HC912B32 variant.

#include <ioa4.h> /* ioa4.h contains symbolic names for the A4 memory-mapped locations.
The base address for the I/O registers defaults to 0x0000 */

```
/*    IO DEFINITIONS FOR MC68HC12A4
 *    Copyright (c) 1996 by COSMIC Software
 */
#define _BASE     0
#define _IO(x)    @_BASE+x
#if _BASE == 0
#define _PORT     @dir
#else
#define _PORT
#endif

_PORT volatile    char PORTA  _IO(0x00);  /* port A */
_PORT volatile    char PORTB  _IO(0x01);  /* port B */
_PORT volatile    char DDRA   _IO(0x02);  /* data direction port A */
_PORT volatile    char DDRB   _IO(0x03);  /* data direction port B */
_PORT volatile    char PORTC  _IO(0x04);  /* port C */
_PORT volatile    char PORTD  _IO(0x05);  /* port D */
_PORT volatile    char DDRC   _IO(0x06);  /* data direction port C */
_PORT volatile    char DDRD   _IO(0x07);  /* data direction port D */
_PORT volatile    char PORTE  _IO(0x08);  /* port E */
.. etc
```

#include <iob32.h> /* iob32.h contains symbolic names for the B32 memory-mapped locations.
The base address for the I/O registers defaults to 0x0000 */

```
/*    IO DEFINITIONS FOR MC68HC12B32
 *    Copyright (c) 1997 by COSMIC Software
 */
#define _BASE     0
#define _IO(x)    @_BASE+x
#if _BASE == 0
#define _PORT     @dir
#else
#define _PORT
#endif

_PORT volatile    char PORTA  _IO(0x00);  /* port A */
_PORT volatile    char PORTB  _IO(0x01);  /* port B */
_PORT volatile    char DDRA   _IO(0x02);  /* data direction port A */
_PORT volatile    char DDRB   _IO(0x03);  /* data direction port B */
_PORT volatile    char PORTE  _IO(0x08);  /* port E */
_PORT volatile    char DDRE   _IO(0x09);  /* data direction port E */
_PORT volatile    char PEAR   _IO(0x0a);  /* port E assignment register */
_PORT volatile    char MODE   _IO(0x0b);  /* mode register */
_PORT volatile    char PUCR   _IO(0x0c);  /* pull-up control register */
_PORT volatile    char RDRIV  _IO(0x0d);  /* reduced drive of I/O lines */
..etc
```

If you have re-mapped the I/O register block to another address, say 0x2000, you should modify the value of _BASE in the appropriate header file, *ioa4.h* or *iob32.h*, to, in this example, address 0x2000.

The following simple example shows code generation of 68HC12 bit instructions

```
1              ; C Compiler for MC68HC12 [COSMIC Software]
2              ; Version V4.1f - 03 Nov 1997
```

```
3                       ; 1 /* Here is a very simple example showing how the HC12 compiler uses the
3                       ; 2    bit instructions:
3                       ; 3 */
3                       ; 4
3                       ; 5 @dir char byte;     /* @dir forces direct page addressing */
3                       ; 6
3                       ; 7 @dir struct {       /* ditto */
3                       ; 8         char b0:1;
3                       ; 9         char b1:1;
3                       ; 10        } bit;
3                       ; 11
3                       ; 12 void f(void)
3                       ; 13        {
4  0000                   _f:
6                       ; 14        if (byte & 4)           /* bit test */
7  0000 4f010403        brclr      _byte,4,L3
8                       ; 15                bit.b0 = 0;     /* clear bit */
9  0004 4d0001          bclr       _bit,1
10 0007                 L3:
11                      ; 16        if (!(byte & 0x80))     /* bit test */
12 0007 4e018003                    brset      _byte,128,L5
13                      ; 17                bit.b1 = 1;     /* set bit */
14 000b 4c0002          bset       _bit,2
15 000e                 L5:
16                      ; 18        if (bit.b0)             /* bit test */
17 000e 4f000103        brclr      _bit,1,L7
18                      ; 19                byte &= ~1;     /* bit clear */
19 0012 4d0101          bclr       _byte,1
20 0015                 L7:
21                      ; 20        if (!bit.b1)            /* bit test */
22 0015 4e000203                    brset      _bit,2,L11
23                      ; 21                byte |= 2;      /* bit set */
24 0019 4c0102          bset       _byte,2
25 001c                 L11:
26                      ; 22        if (bit.b0 == 1)
27 001c 4f000103        brclr      _bit,1,L31
28                      ; 23                bit.b1 = 0;
29 0020 4d0002          bclr       _bit,2
30 0023                 L31:
31                      ; 24        }
32 0023 3d                          rts
33                                  xdef       _f
34                                  bsct
35 0000                 _bit:
36 0000 00                          ds.b       1
37                                  xdef       _bit
38 0001                 _byte:
39 0001 00                          ds.b       1
40                                  xdef       _byte
41                                  end
```

Use of the EMUL/EMULS instructions

The 68HC12 compiler can produce the *emul* and *emuls* instruction, but the C syntax has to be carefully checked. Here is a simple example:

```
                    ; C Compiler for MC68HC12 [COSMIC Software]
                    ; Version V4.1f - 03 Nov 1997
                    ; 1 long l;
                    ; 2 int i,j,k;
                    ; 3
                    ; 4 void main(void)
                    ; 5     {
0000                _main:
                    ; 6
                    ; 7     i = j * k;          /* int = int * int */
0000 fc0002             ldd   _j
0003 fd0000             ldy   _k
0006 13                 emul
0007 7c0004             std   _i
                    ; 8     l = i * j;          /* long = int * int */
000a fd0002             ldy   _j
000d 13                 emul
000e b704               sex   a,d
0010 b701               tfr   a,b
                    ; 9     l = (long) i * j;   /* long = long * int */
0012 fc0004             ldd   _i
0015 fd0002             ldy   _j
0018 1813               emuls
001a 7c0008             std   _l+2
001d 7d0006             sty   _l
                    ; 10 }
0020 3d                 rts
                        xdef  _main
                        switch      .bss
0000                _k:
0000 0000               ds.b  2
                        xdef  _k
0002                _j:
0002 0000               ds.b  2
                        xdef  _j
0004                _i:
0004 0000               ds.b  2
                        xdef  _i
0006                _l:
0006 00000000           ds.b  4
                        xdef  _l
                        end
```

Use of the EMACS instruction

The 68HC12 compiler can produce the *emacs* instruction, but the C syntax has to be carefully checked. Here is a simple example:

```
long lv;
int i1, i2;

f()
        {
        lv += (long)i1 * i2;
        }
```

which produces the following code:

```
 4  0000                _f:
 6                      ; 6     lv += (long)i1 * i2;
 7  0000 ce0002                 ldx     #_i1
 8  0003 cd0000                 ldy     #_i2
 9  0006 18120004               emacs   _lv
10                      ; 7     }
11  000a 3d                         rts
```

You can also expand the += operator and write:

```
lv = lv + (long)i1 * i2;
```

or even apply the (long) cast to i2, or both, but if the cast is applied to the result of the multiplication, such as (long)(i1 * i2), it does not work, because of C evaluation rules. In such a case, the multiplication is done with an *int* resolution, and the 16 bit result is then promoted to a 32 bit value, which does not give the same result of course as a 16x16->32 operation as implemented in the emacs instruction. Note that lv can be an unsigned long, but i1 and i2 have to be signed *ints* (default for a plain *int*), or *char's* (whatever signed or not), or mixed of course.

You have the same constraint if you want to get a full 32 bit result in a long from the multiplication of two ints. Writing:

```
lv = i1 * i2;
```

does a 16x16->16 operation, by using the *emul* instruction (which in fact produces a full 32 bit result), and the 16 bit result is promoted to a long, thus breaking the upper part of the intermediate result.

The proper syntax is:

```
lv = (long)i1 * i2;
```

In such a case, the evaluation rules force the compiler to:

1) promote i1 to a long
2) because now the operation * is between a long and an int, the second int i2 is promoted to a long (this is why one cast is enough on i1 or i2)
3) perform a 32x32->32 operation
4) store the 32 bit result

This sequence is recognized by the compiler as a pattern for optimization, because this operation clearly wants a 32 bit result from the product of two ints, and then the rule allowing a C compiler to shorten the basic operations can be applied, as the final result will always be the same whatever the method used is.

EEPROM Support

If you are using a version of the 68HC12 family, such as the 68HC812A4, that has on-chip EEPROM, the C compiler supports a useful feature to support writing into EEPROM space. If you declare an object as:

```
@eeprom char version[15] = “V4.1C 7/22/97” /* version is array of chars in EEPROM
space */

void main()
      {
      version[4] = ‘D’; /* write a char into EEPROM */
      }
```

The resulting listing file is:

```
 1                ; C Compiler for MC68HC12 [COSMIC Software]
 2                ; Version V4.1f - 03 Nov 1997
 3                .eeprom:        section
 4  0000            _version:
 5  0000 56342e314320       dc.b    "V4.1C 7/22/97",0
 6  000e 00               ds.b    1
 7                ; 1 @eeprom char version[15] = "V4.1C 7/22/97"; /* version is array of chars in EEPROM space *
 7                ; 2
 7                ; 3 void main()
 7                ; 4     {
 8                       switch  .text
 9  0000            _main:
11                ; 5     version[4] = 'D';  /* write a char into EEPROM */
12  0000 c644             ldab    #68
13  0002 cd0004           ldy     #_version+4
14  0005 160000           jsr     c_eewrc
15                ; 6 }
16  0008 3d               rts
17                      xdef    _main
18                      xdef    _version
19                      xref    c_eewrc
20                      end
```

Line 5 of the C source is attempting to write into EEPROM space, which normally requires you to implement a special write sequence to burn the EEPROM. COSMIC C does this automatically for you; the assembly language generated makes a call to c_eewrc which actually writes the character into the EEPROM. You can attach @eeprom to any standard C data type, including complex data types like arrays and structures and the compiler will automatically handle the write sequence for you.

Note that @eeprom data declarations, like for the array version[] in the above example, cause the compiler to generate the data into the .eeprom program section which must be located at link time for the correct address as in:

```
..
+seg .eeprom -b 0x1000 -m4096 # HC812A4 4kb EEPROM is located at address 0x1000
..
```

Note: If you change the default location of the 68HC12 I/O register block from 0x0000 **or** you map the EEPROM from its default address of 0x1000 **or** your system bus-speed is not 8MHz, **and** you are using the @eeprom type qualifier, you also need to modify the file *eeprom.s* which is included in the *libm* source directory that came with the C compiler (library source modules are not installed during compiler installation, so you will have to install directly from the original distribution media). You need to edit some of the assembly language definitions at the top of *eeprom.s*:

```
;       EEPROM WRITE ROUTINES
;       Copyright (c) 1995 by COSMIC Software
;       - eeprom address in Y
;       - value in D and 2,X for longs
;
        xdef    c_eewrc, c_eewrw, c_eewra, c_eewrl
        xdef    _eepera
;
;       the following values have to be modified
;       depending on the processor type and speed
;
EEPROG: equ     $F3             ; control register (change this if EEPROG is mapped to another
address)
EBASE:  equ     $1000           ; eeprom starting address (change this if the EEPROM is located at a
                                ; different base address)
TWAIT:  equ     20000           ; wait value for 10ms @8Mhz (change this for different bus speeds)
;
;       program one word
..
..
```

Some example declarations of data that is located in EEPROM or which points at data in EEPROM::

Example 1

```
@eeprom char c;                 /* c is a char-sized object in EEPROM */
@eeprom int i;          /* i is an int-sized object in EEPROM */
@eeprom int *ptr_eeint;/* ptr_eeint is a pointer to an int-sized object in EEPROM */
int * @eeprom eeptr_int;        /* eeptr_int is located in EEPROM and is a pointer to an int-sized
object */
@eeprom int * @eeprom eeptr_eeint;    /* eeptr_eeint is located in EEPOM and is a pointer to an
                                        int-sized object which is also located in EEPROM (!!)
```

Apart from ptr_eeint, data declared in Example 1 above will be allocated space in the .eeprom program section which can be located at 0xb600 at link time; ptr_eeint is allocated space in the .bss section.

Example 2

You can also use the C compiler's absolute addressing capability to declare each variable to an explicit address. Note that in this case, the compiler generates code to reference the absolute addresses directly but that no space reservations are made for the @eeprom data:

```
@eeprom char c @0x1000;       /* c is in EEPROM at address 0x1000 */
@eeprom int i @0x1001;                /* i is in EEPROM at address 0x1001 */
@eeprom int *ptr_eeint @0x1003; /* ptr_eeint points at an int-sized object at 0x1003 in EEPROM */
int * @eeprom eeptr_int @0x1005; /* eeptr_int is located in EEPROM at address 0x1005 */
@eeprom int * @eeprom eeptr_eeint @0x1007;/* eeptr_eeint is located in EEPOM at address 0x1007 */

void main(void)
{
        c = 'a';
        i = 256;
        *ptr_eeint = 256;
        eeptr_int = (int *)0x1001;
        *eeptr_int = 512;                /* i = 512 */
        eeptr_eeint = (int *)0x1040;
        *eeptr_eeint = 1024;      /* write 1024 as an int into address 0x1040 */
}
```

The resulting listing file is:

```
  1                 ; C Compiler for MC68HC12 [COSMIC Software]
  2                 ; Version V4.1f - 03 Nov 1997
 15                 ; 1 @eeprom char c @0x1000;/* c is in EEPROM at address 0x1000 */
 15                 ; 2 @eeprom int i @0x1001;              /* i is in EEPROM at address 0x1001 */
 15                 ; 3 @eeprom int *ptr_eeint @0x1003; /* ptr_eeint points at an int-sized object at 0x1003 in
EEPROM */
 15                 ; 4 int * @eeprom eeptr_int @0x1005; /* eeptr_int is located in EEPROM at address 0x1005 */
 15                 ; 5 @eeprom int * @eeprom eeptr_eeint @0x1007;/* eeptr_eeint is located in EEPOM at
address 0x1007 */
 15                 ; 6
 15                 ; 7 void main(void)
 15                 ; 8 {
 16 0000               _main:
 18                 ; 9       c = 'a';
 19 0000 c661             ldab      #97
 20 0002 cd1000           ldy       #_c
 21 0005 160000           jsr       c_eewrc
 22                 ; 10      i = 256;
 23 0008 cc0100           ldd       #256
 24 000b cd1001           ldy       #_i
 25 000e 160000           jsr       c_eewrw
 26                 ; 11      *ptr_eeint = 256;
 27 0011 fd1003           ldy       _ptr_eeint
 28 0014 160000           jsr       c_eewrw
 29                 ; 12      eeptr_int = (int *)0x1001;
 30 0017 cc1001           ldd       #4097
 31 001a cd1005           ldy       #_eeptr_int
 32 001d 160000           jsr       c_eewrw
 33                 ; 13      *eeptr_int = 512;                /* i = 512 */
 34 0020 cc0200           ldd       #512
 35 0023 fd1005           ldy       _eeptr_int
 36 0026 6c40             std       0,y
 37                 ; 14      eeptr_eeint = (int *)0x1040;
 38 0028 cc1040           ldd       #4160
 39 002b cd1007           ldy       #_eeptr_eeint
```

```
 40  002e 160000        jsr     c_eewrw
 41                 ; 15    *eeptr_eeint = 1024;      /* write 1024 as an int into address 0x1040 */
 42  0031 cc0400        ldd     #1024
 43  0034 fd1007        ldy     _eeptr_eeint
 44  0037 160000        jsr     c_eewrw
 45                 ; 16 }
 46  003a 3d            rts
 47                     xdef    _main
 48                     xref    c_eewrw
 49                     xref    c_eewrc
 50                     end
```

In Example 2 the addresses for reads and writes are resolved by the compiler before link time, so you need to reserve space for the @eeprom data at link time to prevent the linker from using the EEPROM addresses for ordinary variables. Declarations that use absolute addressing cannot be externed in other source modules, but may be duplicated and included as a header file. To reserve space in your link command file:

```
..
+seg .data -b0x1000 +spc .data=4096 #reserve 4096 bytes at 0x1000 for @eeprom data
..
```

Bank-Switching Support

The @far type modifier is used throughout COSMIC's suite of C compilers to uniformly represent an object that requires greater than 16-bit i.e. 32-bit addressing. The core 68HC12 architecture has 16 address lines, which gives it a *logical* address space of 64kb, so the @far mechanism allows addressing of expanded *physical* memory which may extend well beyond the 64kb logical address space. The expanded memory system employed by some 68HC12 devices uses fast on-chip logic to implement a transparent bank-switching scheme, which improves code efficiency and reduces system complexity. MCUs with expanded memory treat 16Kbytes of memory space from 0x8000 to 0xBFFF as a program memory window. Expanded memory devices also have an 8-bit program page register (PPAGE), which allows up to 256 16-Kbyte program memory pages to be switched into and out of the program memory window. This provides for up to 4 Mbytes of paged program memory.

Note that the C compiler supports bank-switching of code only; it does not directly support bank-switching of data i.e. @far cannot be used on data declarations.

There are two main considerations when dealing with bank switching:

1. How to use the C compiler to make use of the bank-switching scheme. By default the compiler supports the 68HC12A4 variant,
2. How to set up the linker to support the bank-switching scheme.

Compiler Support for Bank-Switching

Let's compile a C module that contains a call to a bank switched function to examine how the compiler handles such functions:

```
extern void @far putstr(char *str); /* declare putstr() as a bank-switched function */

void main(void)
      {
      putstr("hello, world\n");
}
```

Notice that the function *declaration* for putstr() declares putstr() as a bank-switched function using the @far modifier; when putstr() is called from main(), the compiler will generate a different calling sequence, using the *call* instruction, from the normal calling sequence. Notice also that the called function, putstr() in this example, must be compiled as an @far function, so that the correct *rtc* instruction is used to return from the call.

The *definition* for putstr() is:

```
void putstr(char *str)
      {

      while (*str != '\0')
            {
            outch(*str);
            str++;
      }
}
```

Notice that the *definition* of putstr() is a standard C definition. You can compile this module using the +modf compile-time option to force the compiler to generate code for putstr() as an @far function.

The code for outch.c is:

```
#include <ioa4.h>
#define TDRE 0x80

void outch(char c)
      {
      while (!(SC1SR1 & TDRE))      /* wait for READY */
            ;
      SC1DRL = c;             /* send it */
      }
```

Now compile main.c:

```
> cx6812 -l -gf main.c
```

The resulting listing file (main.ls) is:

```
   1                    ; C Compiler for MC68HC12 [COSMIC Software]
   2                    ; Version V4.1f - 03 Nov 1997
   3                    ; 1 extern void @far putstr(char *str);  /* declare
putstr() as a bank-switched function */
   3                    ; 2
   3                    ; 3 void main(void)
   3                    ; 4       {
   4  0000              _main:
   6                    ; 5  putstr("hello, world\n");
   7  0000 cc0000            ldd   #L3
   8  0003 4a000000          call  f_putstr
  10                    ; 6 }
```

```
 11  0007 3d                rts
 12                         xdef  _main
 13                         xref  f_putstr
 14                    .const:    section
 15  0000              L3:
 16  0000 68656c6c6f2c      dc.b  "hello, world",10,0
 17                         end
```

Notice the call f_putstr generates a 24-bit address for the call instruction and that the function name has been prefixed with “f_” to indicate it is an @far function call.

Now we have to compile putstr.c as an @far function:

```
> cx6812 -l -gf +modf putstr.c
```

The resulting listing file (putstr.ls) is:

```
  1                    ; C Compiler for MC68HC12 [COSMIC Software]
  2                    ; Version V4.1f - 03 Nov 1997
  3                         xref  f_outch
  4                    ; 1 void putstr(char *str)
  4                    ; 2  {
  5  0000              f_putstr:
  6  0000 3b                pshd
  7       00000000     OFST:      set   0
  9  0001 b746              tfr   d,y
 10  0003 200e              bra   L5
 11  0005              L3:
 12                    ; 3
 12                    ; 4  while (*str != '\0')
 12                    ; 5        {
 12                    ; 6        outch(*str);
 13  0005 e6f30000          ldab  [OFST+0,s]
 14  0009 87                clra
 15  000a 4a000000          call  f_outch
 17                    ; 7        str++;
 18  000e ed80              ldy   OFST+0,s
 19  0010 02                iny
 20  0011 6d80              sty   OFST+0,s
 21  0013              L5:
 22                    ; 4  while (*str != '\0')
 23  0013 e640              ldab  0,y
 24  0015 26ee              bne   L3
 25                    ; 8  }
 25                    ; 9 }
 26  0017 31                puly
 27  0018 0a                rtc
 28                         xdef  f_putstr
 29                         end
```

Note that the “f_” string has been prefixed to the function names and that an *rtc* instruction terminates f_putstr().

If, however, +modf was not specified at compile time as in:

```
> cx6812 -l -gf  putstr.c
```

The resulting listing file is different:

```
  1                     ; C Compiler for MC68HC12 [COSMIC Software]
  2                     ; Version V4.1f - 03 Nov 1997
  3                          xref  _outch
  4                     ; 1 void putstr(char *str)
  4                     ; 2  {
  5  0000               _putstr:
  6  0000 3b                 pshd
  7       00000000      OFST:     set   0
  9  0001 b746                tfr   d,y
 10  0003 200d                bra   L5
 11  0005               L3:
 12                     ; 3
 12                     ; 4  while (*str != '\0')
 12                     ; 5        {
 12                     ; 6        outch(*str);
 13  0005 e6f30000            ldab  [OFST+0,s]
 14  0009 87                  clra
 15  000a 160000              jsr   _outch
 17                     ; 7        str++;
 18  000d ed80                ldy   OFST+0,s
 19  000f 02                  iny
 20  0010 6d80                sty   OFST+0,s
 21  0012               L5:
 22                     ; 4  while (*str != '\0')
 23  0012 e640                ldab  0,y
 24  0014 26ef                bne   L3
 25                     ; 8  }
 25                     ; 9 }
 26  0016 31                  puly
 27  0017 3d                  rts
 28                          xdef  _putstr
 29                          end
```

There is no “f_” prefix added to function names – just the standard “_” prefix character and putstr() terminates with an *rts* instruction.

The standard run-time libraries (libd.h12, libf.h12, libi.h12, libm.h12) are not built as @far libraries. The machine library, libm.h12, should only be linked into a root segment and should not be placed in paged memory; common library functions should also only be located in the root segment. To create libraries of banked library functions, you need to recompile the sources with the +modf option specified.

Linker Support for Bank-Switching

The linker provides some important link-time options required for bank switching support. The options are:

Global Command Line Options:

-bs# which sets the window size to 2**# (2 to the power #). When # has the value 13, this yields a value of 8 Kbytes for the window size; a value of 14 yields a 16 Kbytes bank size; a value of 15 yields a 32 Kbytes bank size. The value of # for the 68HC12 should be set to 14.

Segment Control Options:

-b## set the physical start address of the segment to the 32-bit address ##.

-m# set the maximum size of each banked segment.

-o# set the logical start address of the segment to the 16-bit address #. This address is the starting

address of the window.

Example

Assume you need a root (non-banked) segment at 0xC000 and you have an 16 Kbyte window origined at 0x8000, so the window appears at logical addresses 0x8000 to 0xBFFF – any addresses seen in this range will address banked external memory. To keep things simple, let's say you have one external 16 Kbyte bank at physical address 0x20000, although in a real banked system it is likely you will employ multiple banks;
Let's say you only want the function putstr() placed in banked memory.

The linker command file, *banked.lkf*, to do this follows:

```
#          LINK COMMAND FILE FOR BANKED EXAMPLE PROGRAM
#          Copyright (c) 1991, 1995 by COSMIC Software (France)
#
# first link the root (non-banked) segment
+seg .text -b 0xC000 -n .EPROM# Root segment is located at 0xC000
+seg .const  -a .EPROM          # constants follow code
+seg .data  -b 0x800            # initialized data is located at 0x800
crtsi.o                         # startup file goes in root segment
main.o                          # so does main C function (interrupt handlers also have to be in root)
outch.o                         # and the character output routine
c:/c/6811/lib/libi.h11          # so does the integer C run-time library
c:/c/6811/lib/libm.h11          # and C machine assist library
#
# now link banked segments
+seg .text -b0x20000 -o0x8000 -m0x4000 -nBANK1 # bank is at physical location 0x20000; window is at 0x8000
putstr.o                        # module containing banked function
#
+seg .const -b0xFFC0 -nvectors# vectors start address
vector.o                              # interrupt vector file
+def __memory=@.bss            # symbol used by library
+def __stack=0xBFF             # set stack at 0xBFF initially
```

The linker command line follows:

```
> clnk -o banked.h12 -bs14 -mbanked.map banked.lkf
> type banked.map
```

Map of banked.h12 from link file banked.lkf

Segments:

```
start 0000c000 end 0000c0a6 length   166 segment .EPROM
start 0000c0a9 end 0000c0bb length    18 segment .const
start 00000800 end 00000800 length     0 segment .data, initialized
start 00000800 end 00000802 length     2 segment .bss
start 00020000 end 00020018 length    24 segment BANK1
start 0000ffc0 end 0000fff2 length    50 segment vectors
start 0000c0a6 end 0000c0a9 length     3 segment .init
```

Modules:

```
crtsi.o:
start 0000c000 end 0000c02e length    46 section .EPROM
```

start 00000800 end 00000800 length 0 section .bss

main.o:
start 0000c02e end 0000c040 length 18 section .EPROM
start 0000c0a9 end 0000c0bb length 18 section .const

outch.o:
start 0000c040 end 0000c047 length 7 section .EPROM

(c:/c/6811/lib/libm.h11)msp.o:
start 0000c047 end 0000c05c length 21 section .EPROM

(c:/c/6811/lib/libm.h11)wcalk.o:
start 0000c05c end 0000c0a6 length 74 section .EPROM
start 00000800 end 00000802 length 2 section .bss

putstr.o:
start 00020000 end 00020018 length 24 section BANK1

vector.o:
start 0000ffc0 end 0000fff2 length 50 section vectors

Stack usage:

_main > 10 (2)
_outch 4 (4)
_putstr 8 (4)

Bank Packing Utility

The bank packing utility program cbank is intended to aid the user, who is using bank switching, to optimize bank filling. Suppose you have an application with two banks and a list of object files. In order to create a linker command file, you need to start the first bank, then specify the objects for the first bank, then open the second bank and fill it with the remaining object files. Where do you decide to open the second bank? Mainly when the first bank is filled, but there is no easy way to know when this happens, unless you link your whole application and get an error message from the linker when the first bank overflows. Also, as your application evolves, some object files will grow and will reach a size where your bank design no longer works. You will then have to modify it, by moving one or more objects from the first to the second bank; this process can be cumbersome and error prone.

To avoid such a process during application design, the COSMIC linker allows the -w option which allows it to automatically switch to the next bank when the current bank overflows, assuming that the available banks are contiguous in memory. But, when the linker detects an overflow, it does not reorganize the object files already linked, which means there will be a 'hole' left in the current bank which is unfilled.

The cbank utility solves this problem by creating a (subset of) linker command file with the object files ordered so that the linker will optimize memory usage and minimize the amount of unused memory, thus minimizing the number of necessary banks. In order to work, you first compile all your code down to object files, with all function calls declared as @far (bank-switched) calls, because at this stage you do not know into which bank they will be allocated. Once this is done, you create a file containing all the object file names and pass this file, as an argument, to the cbank utility. You also must specify the bank size, and you can specify which program section is to be packed (by default, cbank uses the .text section). Cbank reads all the object files, looking at their sizes, and will sort the objects in order to produce, as its output, a list of objects files sorted so as to give the best fit, for your bank-switched design.

At this point you may decide to let the linker work on the sorted list of object files in conjunction with the -w option, or you may ask cbank to also include the segment directives in its output file. In this latter case, you need to specify, at the top of cbank's input file, the list of available banks. Cbank will move those directives to the correct place in its output file.

Cbank's output file can be read directly by your link command file using the +inc directive, which acts just like an 'include' directive to the linker, loading the contents of the file into the linking process.

Most of the time, a banked application has one or more non-banked segments containing the C run-time startup code (*crtsi.o*), interrupt handlers and common libraries. So the input file for cbank contains only the object files which are to be loaded into banked memory. Note that there is also a problem with constant data (which is general to any bank-switching scheme); if constants are output to a separate section, they should be linked in any non-banked area to allow direct access from anywhere. If constants are generated into the .text section (see +nocst compile-time option), this means that they will be loaded with the code into a bank, and any *const* variable becomes 'static' because it cannot be accessed outside the allocated bank.

To use the cbank utility correctly you must perform the following operations:

1. Set up an input file with the named object files (say a file called *object.txt*)
2. Run cbank to create an output file (say a file called *banks.txt*)

```
cbank -o banks.txt -w0x2000 object.txt
```

3. Insert the resulting file into your linker command file after the first bank opening:

```
+ seg .text -b 0x10000 -o 0x4000 -w 0x2000
+inc banks.txt
```

The options which may be specified to cbank are:

```
usage: cbank [options] <file>
      -m#   maximum available banks
      -n*   name of section to pack
      -o*   output file name
      -w##  bank size
```

The -m options allows a definition of the maximum number of wanted banks. Accurate definition of this option will enhance the efficiency of cbank. Note that if this option is not specified, cbank will compute an upper limit by first filling the banks without packing.

The -n option specifies the name of the section which is to be packed and, by default, this is the .text section. It is not possible to pack several sections together, so the compiler should be used with the +nocst option to force constants into the .text section.

The -o option specifies the output filename for cbank output. If not specified, the result is sent to standard output (your terminal screen).

The -w option specifies the bank size. This information can also be found in the cbank command file, but if both are specified, the cbank utility will use the value specified with -w in the command line. This information must be specified, otherwise cbank will generate an arror message.

The cbank command file starts with a description of the banks and continues with a list of object filenames. A bank description uses the same options as a linker command file:

-o	logical start address of a bank. This can be specified only once as all banks start at the logical address
-w	bank size. This can be specified only once as all banks have the same size. This may be overwritten by the -w## command line option
-b	physical start address of one bank. This can be entered as many times as there are available banks
-p	page value of a bank (if the -bs option is not used). This can be entered as many times as there are available banks
-n	segment name. This can be entered as many times as there are available banks.

Note that there must be the same number of -b, -p and -n entries and that the -p and -n options are not mandatory. Except for the -w option, which requires a numerical value, all other options accept text strings in order to be compatible with the expression syntax of the COSMIC linker.

If at least one -b option is specified, cbank will create the +seg directives between bank definitions and in such a case the -o option is manditory. If no -b option is specified, the result contains only the object filenames separated by comments, and the +seg directives have to be specified in the linker command file with the -w option specified to activate the automatic filling feature of the linker.

Example:

```
# global definitions
#
-w 0x4000               # bank size is 16kb
-o 0x2000               # logical start address
```

```
#
# definition of banks
#
-b 0x2000 -n bank0      # bank 0 parameters
-b 0x10000 -n bank1     # bank 1 parameters
-b 0x14000 -n bank2     # bank 2 parameters
-b 0x18000 -n bank3     # bank 3 parameters
-b 0x1c000 -n bank4     # bank 4 parameters
#
# object files
#
file1.o file2.o file3.o
file4.o file5.o file6.o
```

The resultant file will contain, after the +seg directives or after the comment separated list of filenames, a description of the bank filling, using the syntax (*used/available*), where *used* is the number of bytes used in the bank and *available* is the bank capacity.

In-line Assembler Statements

The C compiler supports two methods for in-lining assembly language statements into C source code.

Method 1 (only available in V4.1x releases)
The first method uses preprocessor directives to enclose assembly language instructions. This is the most convenient method to use if large sequences of code are to be in-lined, but there is no provision to interface with C data.

The compiler accepts the following sequences to start and finish assembly language blocks of code:

```
#pragma asm       /* start assembler block */
#pragma endasm    /* finish assembler block */
```

The following sequences are also accepted:

```
#asm              /* start assembler block */
#endasm           /* finish assembler block */
```

Assembly language inserts may be located anywhere, inside or outside of a C function. Outside a C function, it behaves syntactically as a C declaration, which means that an assembler block cannot split a C declaration. When used inside a C function, it behaves syntactically as one C instruction. This means that there is no trailing semicolon at the end and no need for enclosing braces. This also implies that an assembly block cannot split a C instruction or expression.

Example:

```
#pragma asm
tpa
#pragma endasm
or
#asm
tpa
#endasm
```

Method 2.
The _asm() method only works inside a C function but it acts just like a C function, so it can be used in expressions, it can pass arguments and it can return a value, provided the assembly language code follows

the C compiler's function return value conventions (see C Compiler documentation for function calling conventions). The syntax is:

```
_asm("string_constant", arguments…);
```

where "string_constant" is the assembly language code to be embedded in your C code and *arguments* follow the standard C rules for passing arguments. "string_constant" must be shorter than 255 characters – if you need to insert longer assembly language code sequences you will have to split your input among several calls to _asm(). Arguments follow the C compiler conventions for passing arguments, where the first argument is passed in register D if it is int sized or smaller, and subsequent arguments are passed on the stack. This means that C data local to a function can also be passed as an argument into the assembly language code for easy access to local C data.

Example: to produce the following assembly language sequence:

```
txs
jsr   _main
```

you would write:

```
_asm("txs\n jsr   _main\n");
```

Example: to transfer a copy of the conditions codes from a global C variable named "varcc" to the ccr register:

```
 1                      ; C Compiler for MC68HC12 [COSMIC Software]
 2                      ; Version V4.1f - 03 Nov 1997
 3                      ; 1 unsigned char varcc;
 3                      ; 2
 3                      ; 3 main() {
 4  0000                _main:
 6                      ; 4     _asm("tap\n", varcc);
 7  0000 f60000              ldab  _varcc
 8  0003 87                  clra
 9  0004 b702                tap
10                      ; 5 }
11  0006 3d                  rts
12                           xdef  _main
13                           switch      .bss
14  0000                _varcc:
15  0000 00                  ds.b  1
16                           xdef  _varcc
17                           end
```

Example: to test the overflow bit:

```
 1                      ; C Compiler for MC68HC12 [COSMIC Software]
 2                      ; Version V4.1f - 03 Nov 1997
 3                      ; 1 main() {
 4  0000                _main:
 6                      ; 2     if (_asm("tpa\n") & 0x80)
 7  0000 b720                tpa
 8  0002 c580                bitb  #128
 9                      ; 3         ;
 9                      ; 4 }
10  0004 3d                  rts
11                           xdef  _main
12                           end
```

Example: to pass local variable b into assembly language code, the arguments are passed first, then the assembler code sequences are in-lined:

```
  1                     ; C Compiler for MC68HC12 [COSMIC Software]
  2                     ; Version V4.1f - 03 Nov 1997
  3                     ; 1 main() {
  4  0000               _main:
  5  0000 3b                 pshd
  6       00000002      OFST:     set   2
  8                     ; 2     volatile char a,b;
  8                     ; 3
  8                     ; 4     b = 2;
  9  0001 c602                ldab  #2
 10  0003 6b80                stab  OFST-2,s
 11                    ; 5     a = _asm("incb\n",b);   /* a = 3 */
 12  0005 e680                ldab  OFST-2,s
 13  0007 52                  incb
 14  0008 6b81                stab  OFST-1,s
 15                    ; 6 }
 16  000a 31                  puly
 17  000b 3d                  rts
 18                           xdef  _main
 19                           end
```

With both methods, the assembler source code is added “as is” to the C code during compilation. The C compiler optimizer does not modify the specified instructions, unless the -a option is specified to the C code generator (***cg6812***). The assembly language instructions may be specified in upper or lower case letters and may include comments. You cannot, however, specify an assembler-level label and an instruction on the same line i.e. labels must be entered on a line by themselves.

Interrupt Handlers at the C Level

The compiler supports function definitions for C functions that service interrupts using the special identifier @interrupt. This causes the compiler to generate an *rti* instruction instead of an *rts* instruction when the function returns. The use of C functions as interrupt service routines does not require the user to do anything special, such as saving other CPU registers.

@interrupt functions should only be called from interrupts, however, and not directly from user application code. Most interrupt functions do not take function arguments and do not return a value, so a typical definition is:

```
@interrupt void isr(void)
      {
      <body of function>
}
```

Example:

The example code below gives a simple example of the use of a C interrupt function, recept(), which is tied to the interrupt vector for the SCI receive interrupt – interrupt vectors are defined in a separate file vector.c not shown in this example. When an SCI receive interrupt occurs, recept() is called to store the character into a buffer, defined to be a 512 byte array of char. Recept() signals receipt of a char to the C standard character receive function getch(), by incrementing the buffer write pointer, ptecr. The main() function sets up the SCI parameters and starts an endless loop of outch(getch()) which receives and transmits characters from/to the SCI.

```
#include <ioa4.h>       /* header file with 68HC812A4 I/O register block definitions */

#define SIZE    512     /* buffer size */
#define TDRE    0x80    /* transmit ready bit */

/*      Authorize interrupts.
 */
#define cli()   _asm("andcc #$EF\n")

/*      Some variables.
 */
char buffer[SIZE];      /* buffer used to store characters received from SCI */
char *ptlec;            /* read pointer */
char *ptecr;            /* write pointer */

/* Main function. Sets up the SCI and starts an infinite loop of SCI receive transmit.
 */
void main(void)
        {
        ptecr = ptlec = buffer;         /* initialize buffer pointers */
        SC1BDL = 55;                    /* initialize SCI */
        SC1CR2 = 0x2c;                  /* parameters for interrupt */
        cli();                          /* authorize interrupts */
        for (;;)                        /* loop */
                outch(getch());         /* get and put a char */
        }


/* Standard C character receive function. Loops until a character is received.
 */
char getch(void)
        {
        char c;                         /* character to be returned */

        while (ptlec == ptecr)          /* are the buffer pointers equal? */
                ;                       /* yes; this means recept() has not received a new
                                           character from the SCI, so just loop */
        c = *ptlec++;                   /* no; get the received char pointed at by ptlec and
                                           increment the buffer read pointer */
        if (ptlec >= &buffer[SIZE])     /* check for buffer read overflow */
                ptlec = buffer;
        return (c);
        }

/*      Send a char to the SCI.
 */
void outch(char c)
        {
        while (!(SC1SR1 & TDRE))                /* wait for READY */
                ;
        SC1DRL = c;                     /* send it */
        }

/* Interrupt handler. This routine is called on SCI interrupt. It puts the received char into buffer
```

```
   and signals to getch() that a character has been received by incrementing the buffer write
   pointer, ptecr. It also checks for write overflow of the buffer.
*/

@interrupt void recept(void)
        {
        SC1SR1;                         /* clear interrupt */
        *ptecr++ = SC1DRL;      /* get the char from SCI Data Register, store it in buffer
                                   and increment the buffer write pointer */
        if (ptecr >= &buffer[SIZE])     /* check for write buffer overflow */
                ptecr = buffer;
        }
```

The resultant output listing file from the C compiler is as follows:

```
  1                 ; C Compiler for MC68HC12 [COSMIC Software]
  2                 ; Version V4.1f - 03 Nov 1997
265                         xref    _getch
266                         xref    _outch
267                   ; 1 #include <ioa4.h>  /* header file with 68HC812A4 I/O register block definitions */
267                   ; 2
267                   ; 3 #define SIZE        512     /* buffer size */
267                   ; 4 #define TDRE        0x80    /* transmit ready bit */
267                   ; 5
267                   ; 6 /*  Authorize interrupts.
267                   ; 7  */
267                   ; 8 #define cli()           _asm("andcc #$EF\n")
267                   ; 9
267                   ; 10 /* Some variables.
267                   ; 11  */
267                   ; 12 char buffer[SIZE];/* buffer used to store characters received from SCI */
267                   ; 13 char *ptlec;                   /* read pointer */
267                   ; 14 char *ptecr;                   /* write pointer */
267                   ; 15
267                   ; 16 /* Main function. Sets up the SCI and starts an infinite loop of SCI receive transmit.
267                   ; 17  */
267                   ; 18 void main(void)
267                   ; 19    {
268 0000                _main:
270                   ; 20    ptecr = ptlec = buffer;             /* initialize buffer pointers */
271 0000 cc0004              ldd     #_buffer
272 0003 7c0002              std     _ptlec
273 0006 7c0000              std     _ptecr
274                   ; 21    SC1BDL = 55;                        /* initialize SCI */
275 0009 c637                ldab    #55
276 000b 5bc9                stab    _SC1BDL
277                   ; 22    SC1CR2 = 0x2c;                      /* parameters for interrupt */
278 000d c62c                ldab    #44
279 000f 5bcb                stab    _SC1CR2
280                   ; 23    cli();                              /* authorize interrupts */
281 0011 10ef                andcc   #$EF
282 0013              L3:
283                   ; 24    for (;;)                            /* loop */
283                   ; 25            outch(getch());             /* get and put a char */
284 0013 0704                jsr     _getch
```

```
 286  0015 0719          jsr       _outch
 289  0017 20fa          bra       L3
 290                  ; 26   }
 290                  ; 27
 290                  ; 28
 290                  ; 29 /* Standard C character receive function. Loops until a character is received.
 290                  ; 30  */
 290                  ; 31 char getch(void)
 290                  ; 32   {
 291  0019              _getch:
 292      00000001     OFST:    set       1
 294  0019 fd0002        ldy       _ptlec
 295  001c              L11:
 296                  ; 33   char c;                          /* character to be returned */
 296                  ; 34
 296                  ; 35   while (ptlec == ptecr)           /* are the buffer pointers equal? */
 297  001c bd0000        cpy       _ptecr
 298  001f 27fb          beq       L11
 299                  ; 36             ;                     /* yes; this means recept() has not received a new
 299                  ; 37                                                 character from the SCI, so just
loop */
 299                  ; 38   c = *ptlec++;                    /* no; get the received char pointed at by ptlec and
 300  0021 e670          ldab      1,y+
 301                  ; 39                                       increment the buffer read pointer */
 301                  ; 40   if (ptlec >= &buffer[SIZE])      /* check for buffer read overflow */
 302  0023 8d0204        cpy       #_buffer+512
 303  0026 2503          blo       L51
 304                  ; 41             ptlec = buffer;
 305  0028 cd0004        ldy       #_buffer
 306  002b              L51:
 307  002b 7d0002        sty       _ptlec
 308                  ; 42   return (c);
 309  002e d7            tstb
 311  002f 3d            rts
 312                  ; 43   }
 312                  ; 44
 312                  ; 45 /* Send a char to the SCI.
 312                  ; 46  */
 312                  ; 47 void outch(char c)
 312                  ; 48   {
 313  0030              _outch:
 314      00000000     OFST:    set       0
 316  0030              L12:
 317                  ; 49   while (!(SC1SR1 & TDRE))              /* wait for READY */
 318  0030 4fcc80fc      brclr     _SC1SR1,128,L12
 319                  ; 50             ;
 319                  ; 51   SC1DRL = c;                      /* send it */
 320  0034 5bcf          stab      _SC1DRL
 321                  ; 52   }
 322  0036 3d            rts
 323                  ; 53
 323                  ; 54 /* Interrupt handler. This routine is called on SCI interrupt. It puts the received char into buffer
 323                  ; 55    and signals to getch() that a character has been received by incrementing the buffer write
 323                  ; 56    pointer, ptecr. It also checks for write overflow of the buffer.
 323                  ; 57  */
 323                  ; 58
```

```
323                 ; 59 @interrupt void recept(void)
323                 ; 60   {
324  0037             _recept:
326                 ; 61   SC1SR1;                          /* clear interrupt */
327  0037 d6cc          ldab    _SC1SR1
328                 ; 62   *ptecr++ = SC1DRL;     /* get the char from SCI Data Register, store it in buffer
329  0039 d6cf          ldab    _SC1DRL
330  003b fd0000        ldy     _ptecr
331  003e 6b70          stab    1,y+
332                 ; 63    and increment the buffer write pointer */
332                 ; 64   if (ptecr >= &buffer[SIZE])        /* check for write buffer overflow */
333  0040 8d0204        cpy     #_buffer+512
334  0043 2503          blo     L52
335                 ; 65           ptecr = buffer;
336  0045 cd0004        ldy     #_buffer
337  0048             L52:
338  0048 7d0000        sty     _ptecr
339                 ; 66   }
340  004b 0b            rti
341                    xdef    _recept
342                    xdef    _outch
343                    xdef    _getch
344                    xdef    _main
345                    switch  .bss
346  0000            _ptecr:
347  0000 0000          ds.b    2
348                    xdef    _ptecr
349  0002            _ptlec:
350  0002 0000          ds.b    2
351                    xdef    _ptlec
352  0004            _buffer:
353  0004 000000000000      ds.b    512
354                    xdef    _buffer
355                    end
```

Floating Point Support

The compiler supports single precision (32-bit) and double precision (64-bit) floating point operations. By default, ANSI C dictates that all floating point arithmetic be done at double precision. You can defeat this default behavior using the **+sprec** compile-time flag as described above.

Double precision floating point arithmetic is typically two to three times slower than single precision arithmetic. The following table gives some benchmark information.

SINGLE PRECISION

FLOAT	Typical Timing Range
addition	136 - 220 cycles
subtraction	164 - 248 cycles
multiply	163 - 177 cycles

division	911 - 1120 cycles
sin	2572 - 2658 cycles
cos	2420 - 2500 cycles
sqrt[1]	5795 - 5854 cycles

DOUBLE PRECISION

FLOAT	Typical Timing Range
addition	279 - 461 cycles
subtraction	327 - 509 cycles
multiply	850 - 950 cycles
division	4061 - 4673 cycles
sin	8560 - 9030 cycles
cos	9400 - 9911 cycles
sqrt[1]	20,320 - 23,061 cycles

**all figures quoted are executed machine cycles*

[1] *The timing figures for sqrt() are high due to the casting of integer i to a double (see source below)*

The source code for the program used for the timing is as follows:

```
/* compile with +sprec option to force single precision (32-bit float) arithmetic for float benchmark
   and then without +sprec for double precision (64-bit) arithmetic */

#include <math.h>

#define _PI 3.1415

void main(void)
      {
      double x,y,z;
      double angle = _PI/4;
      int i;

      x = 12345.678912;
      y = 0.987654;

      for (i=0; i < 200; i++)
            {
            z = x + y;       /* add timing – includes time to access x and y and store result in z */
            z = y - x;       /* subtract  – includes time to access x and y and store result in z */
            z = x * y;       /* multiply – includes time to access x and y and store result in z */
            z = y / x;       /* divide – includes time to access x and y and store result in z */
            z = sin(angle);          /* timing includes time to access angle and store result in z */
            z = cos(angle);          /* timing includes time to access angle and store result in z */
            z = sqrt((double) i);    /* timing includes time to access angle and store result in z */
            x -= 10.0;
            y += 10.0;
```

```
            angle += 0.2 / (double) i++;
    }
}
```

Assembler Considerations

This part of the application note is intended to provide users who have existing 68HC12 assembly language code, which assembles with their current assembler, with information about the assembly language syntax accepted by the COSMIC 68HC12 assembler included in the V4.1x COSMIC C cross compiler package. Users should use this note as a quick aid to understanding the amount of effort (if any) in converting existing assembler code to assemble cleanly with the COSMIC 68HC12 assembler.

Invoking ca6812

The COSMIC 6812 assembler, ca6812, is an MCUASM compatible assembler for the 68HC12 family of microcontrollers. It can generate listing (with or without cross-references), error and relocatable object files and accepts the following command-line options:

```
>ca6812

COSMIC Software Macro-Assembler V4.1f
usage: ca6812 [options] files
    -a            absolute assembler
    -b            do not optimize branches
    -c            output cross reference
    -d*>          define symbol=value
    +e*           error file name
    -ff           use formfeed in listing
    -ft           force title in listing
    -f#           fill byte value
    -h*           include header
    -i*>          include path
    -l            output a listing
    +l*           listing file name
    -mi           accept label syntax
    -m            accept old syntax
    -o*           output file name
    -pe           all equates public
    -pl           keep local symbols
    -p            all symbols public
    -v            verbose
    -xp           no path in debug info
    -xx           include full debug info
    -x            include line debug info
```

Language Syntax

ca6812 conforms to the Motorola syntax as described in the document *Assembly Language Input Standard* and consists of lines of text in the form:

[label:][command[operands]] [;comment]

or

; comment

where ‘:’ indicates the end of a label and ‘;’ defines the start of a comment. The end of a line terminates a comment. The *command* field may be an instruction, a directive or a macro call. Instruction mnemonics

and assembler directives may be written in upper or lower case and a source file must end with the **end** directive.

Instructions

ca6812 recognizes the following 68HC11 instructions:

aba	**abx**	**aby**	**adca**	**adcb**	**adda**
addb	**addd**	**anda**	**andb**	**asl**	**asla**
aslb	**asld**	**asr**	**asra**	**asrb**	**bcc**
bclr	**bcs**	**beq**	**bge**	**bgt**	**bhi**
bhs	**bita**	**bitb**	**ble**	**blo**	**bls**
blt	**bmi**	**bne**	**bpl**	**bra**	**brclr**
brn	**brset**	**bset**	**bsr**	**bvc**	**bvs**
cba	**clc**	**cli**	**clr**	**clra**	**clrb**
clv	**cmpa**	**cmpb**	**com**	**coma**	**comb**
cpd	**cpx**	**cpy**	**daa**	**dec**	**deca**
decb	**des**	**dex**	**dey**	**eora**	**eorb**
fdiv	**idiv**	**inc**	**inca**	**incb**	**ins**
inx	**iny**	**jmp**	**jsr**	**ldaa**	**ldab**
ldd	**lds**	**ldx**	**ldy**	**lsl**	**lsla**
lslb	**lsld**	**lsr**	**lsra**	**lsrb**	**lsrd**
mul	**neg**	**nega**	**negb**	**nop**	**oraa**
orab	**psha**	**pshb**	**pshx**	**pshy**	**pula**
pulb	**pulx**	**puly**	**rol**	**rola**	**rolb**
ror	**rora**	**rorb**	**rti**	**rts**	**sba**
sbca	**sbcb**	**sec**	**sei**	**sev**	**staa**
stab	**std**	**stop**	**sts**	**stx**	**sty**
suba	**subb**	**subd**	**swi**	**tab**	**tap**
tba	**test**	**tpa**	**tst**	**tsta**	**tstb**
tsx	**tsy**	**txs**	**tys**	**wai**	**xgdx**
xgdy					

and the following additional 68HC12 instructions:

andcc	**bgnd**	**call**	**cps**	**dbeq**	**dbne**
ediv	**edivs**	**emacs**	**emaxd**	**emaxm**	**emind**
eminm	**emul**	**emuls**	**etbl**	**exg**	**ibeq**
ibne	**idivs**	**lbcc**	**lbcs**	**lbeq**	**lbge**
lbgt	**lbhi**	**lbhs**	**lble**	**lblo**	**lbls**
lblt	**lbmi**	**lbne**	**lbpl**	**lbra**	**lbrn**
lbsr	**lbvc**	**lbvs**	**leas**	**leax**	**leay**
maxa	**maxm**	**mem**	**mina**	**minm**	**movb**
movw	**orcc**	**pshc**	**pshd**	**pulc**	**puld**
rev	**revw**	**rtc**	**sex**	**tbeq**	**tbl**
tbne	**tfr**	**wav**			

Labels

A source line may begin with a label. Some directives require a label on the same line, otherwise this field is optional. A label begins with an alphabetic character, the underscore character '_' or the dot character '.'. It is continued by alphabetic or numeric characters. Labels are case sensitive. The processor register names 'a', 'b', 'x', and 'y' are reserved and cannot be used as labels.

When a label is used inside a macro it may be expanded more than once which will cause failure. To avoid the problem, the special sequence '\@' may be used as a label prefix. This sequence will be replaced by a unique sequence for each macro expansion.

Constants

ca6812 accepts numeric and string constants; numeric constants are expressed in different bases depending on a prefix character as follows:

10	decimal (no prefix)
%1010	binary
@12	octal
$A	hexadecimal

String constants are a series of printable characters between single or double quote characters:

'This is a string'
"This is also a string"

Expressions

Expressions are evaluated to 32-bit precision and operators have the same precedence as in the C language. A special label '*' is used to represent the current location address and when used as the operand of an instruction, it has the value of the program counter before code generation for that instruction. Operators are:

+	addition
-	subtraction (negation)
*	multiplication
/	division
%	remainder (modulus)
&	bitwise AND
|	bitwise OR
^	bitwise EXCLUSIVE OR
~	bitwise complement
<<	left shift
>>	right shift
==	equality
!=	difference
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

&&	logical AND
||	logical OR
!	logical complement

These operators may be applied to constants without restrictions but are restricted when applied to relocatable labels, in which case the addition and subtraction operators only are accepted and only in the following cases:

label + constant
label - constant
label1 - label2

The difference of two relocatable labels is valid only if both symbols are not external symbols and are defined in the same program section.

Additional expressions are:

high(expression)	extract the upper byte of the 16-bit expression
low(expression)	extract the lower byte of the 16-bit expression
page(expression)	extract the page value of the expression

Macro Instructions

A macro begins with a **macro** directive and ends with an **endm** directive as in:

```
signex:     macro       ;sign extend
      clra
      tstb        ;test sign
      bpl   \@pos ;if not negative
      coma        ;invert MSB
\@pos:
      endm        ;end of macro
```

The above macro is named signex. A macro can have up to nine parameters written \1, \2, ... \9 inside the macro body and refers explicitly to the first, second,.... ninth argument in the invocation line, which are placed after the macro name and separated by commas. An argument may be expressed as a string constant if it contains a comma character.

Example:

```
signex:     macro
      clra        ;prepare MSB
      ldab  \1+1  ;load LSB
      bpl   \@pos ;if not negative
      coma        ;invert MSB
\@pos:
      std   \1    ;store MSB
      endm
```

and called:

```
      signex char ;sign extend char
```

The special parameter written \0 is replaced by a numeric value corresponding to the number of arguments actually found on the invokation line.

The directive **mex**it may be used at any time to stop the macro expansion.

A macro call may be used within another macro definition. A macro definition cannot contain another macro definition.

Conditional Directives

The ***if*** directive allows parts of the program to be conditionally assembled depending on a specific condition following the ***if*** directive, which cannot be relocatable and must evaluate to a numeric result. If the condition is false (expression evaluated to zero), the lines following the ***if*** directive are skipped until an ***endif*** or ***else*** directive.

Example:

```
if    offset != 1 ;if offset too large
addptr      offset            ;call a macro
else
inx               ;otherwise inc x reg
endif
```

If the symbol *offset* is not equal to one, the macro *addptr* is expanded with *offset* as an argument, otherwise the *inx* instruction is directly assembled.

Conditional directives may be nested. An ***else*** directive refers to the closest previous ***if*** directive and an ***endif*** refers to the closest previous ***if*** or ***else*** directive.

Includes

The ***include*** directive specifies a file to be included and assembled in place of the ***include*** directive. The file name is written between double quotes, and may be any file describing a file on the host system. If the file cannot be found using the given name, it is searched from all the include paths defined by the **-i** command-line options, and from the paths defined by the environment symbol **CXH6812**, if such a symbol has been defined. The **-h** options can specify a file to be included, which will be included as if the program had an include directive at its very top.

Sections

Code and data can be split into sections using the **section** directive. A section is a set of code or data referenced by a section name; to switch between program sections, the **switch** directive is used:

Example:

```
      .data:      section     ; defines .data section
      .text:      section     ; defines .text section (code)
start:      ldx   #value      ; fills .text section
      jmp   print
      switch      .data ; switch to .data section
value:      dc.b  1,2,3 ; fills .data section
```

The assembler allows up to 255 different program sections and a section name is limited to fifteen characters.

Branch Optimization

Branch instructions are by default automatically optimized to generate the shortest code possible, but this behaviour may be disabled by the **-b** command-line option. This optimization operates on conditional branches, on jumps and jumps to subroutine.

A ***jmp*** or ***jsr*** instruction will be replaced by a ***bra*** or ***bsr*** instruction if the destination address is in the same section as the current one, and if the displacement is in the range allowed by a relative branch. The ***bra*** instruction will be replaced by a single ***jmp*** instruction if it cannot be encoded as a relative branch.

A conditional branch offset is limited to the range [-128,127] and if such an instruction cannot be encoded properly, the assembler will replace it by a sequence containing an inverted branch to the next location followed immediately by a jump to the original target address. The assembler keeps track of the last replacement for each label, so if a long branch has already been expanded for the same label at a location close enough to the current instruction, the target address of the short branch will be changed only to branch on the already existing jump instruction to the specified label.

```
beq   farlabel
```

becomes

```
bne   *+5
jmp   farlabel
```

Old Syntax

The **-m** option allows the assembler to accept old constructs which are now obsolete:

- a comment line may begin with a '*' character
- a label starting in the first column does not need to end with the ':' character
- no error message is issued if an operand of the **dc.b** directive is too large
- the **section** directive accepts numbered sections

Assembler Directives

The following table gives a brief description of the assembler directives supported in ca6812:

Directive Name	Description	Example
align <expression>	**align the next instruction on a given boundary**	**align 3**
base <expression>	**define the default base for numerical constants; <expression> must be one of 2,8,10 or 16**	**base 2**
bsct	**Switch to the predefined .bsect section, also known as the zero page section**	**bsct**
clist [on|off]	**turn listing of conditionally excluded code on or off**	**clist on**
dc[.size] <expression>[,<expression>]	**Allocate and initialize storage for constants.**	**dc 10,'0123456789' dc.b 10,'0123456789' dc.w word dc.l longword**
dcb.<size> <count>,<value>	**Allocate a memory block and initialize storage for constants. The size area is**	**dcb.b 10,5 dcb 10,5**

	<count> of <size> which can be initialized with <value>	**dcb.w 10,5** **dcb.l 10,5**
dlist [on|off]	**Turn listing of debug directives on or off**	**dlist on**
ds[.size] <space>	**allocate storage space for variables. <space> must be an absolute expression. Bytes created are set to the value of the filling byte defined by the -f command-line option**	**ptlec: ds.b 2** **ptecr: ds.b 2** **chrbuf: ds.w 128**
else	**conditional assembly**	
elsec	**alias for else**	
end	**halt assembly**	**end**
endc	**end conditional assembly. Alias for endif**	
endif	**end conditional assembly**	
endm	**end macro definition**	
equ <expression>	**give a permanent value to a symbol**	**false: equ 0** **true equ 1** **tablen: equ tabfin - tabsta** **nul equ $0**
even	**assemble next byte at the next even address relative to the start of a section**	**even**
fail "string"	**generate error message**	**fail "Value too large"**
if <expression> or if <expression> **instructions instructions** **endif else** **instructions** **endif**	**Conditional assembly**	
ifc <string1>,<string2> **instructions** **elsec** **instructions** **endc**	**ifc, else and endc directives allow conditional assembly. If <string1> and <string2> are equal, the following instructions are assembled up to the next matching endc or elsec directive**	**ifc "hello", \2** **ldab #45** **elsec** **ldab #0** **endc**
ifeq <expression> **instructions** **elsec** **instructions** **endc**	**conditional assembly; test for equality to zero**	**ifeq offset** **tsta** **elsec** **adda #offset** **endc**
ifge <expression> **instructions** **elsec** **instructions** **endc**	**conditional assembly; test for greater than or equal to zero**	**ifge offset - 127** **addptr offset** **elsec** **inx** **endc**
ifgt <expression> **instructions** **elsec** **instructions** **endc**	**conditional assembly; test for greater than zero**	
ifle <expression> **instructions** **elsec** **instructions** **endc**	**conditional assembly; test for less than or equal to zero**	
iflt <expression> **instructions**	**conditional assembly; test for less than zero**	

elsec instructions endc		
ifne <expression> instructions elsec instructions endc	**conditional assembly; test for not equal to zero**	**ifne offset adda #offset elsec tsta endc**
ifnc <string1>,<string2> instructions elsec instructions endc	**conditional assembly; If <string1> and <string2> are unequal, the following instructions are assembled up to the next matching endc or elsec directive**	**ifnc "hello", \2 addptr offset elsec inx endc**
include	**include text from another text file**	**include "datstr" include "matmac"**
list	**turn on listing during assembly**	**list**
label: macro <macro body> endm	**define a macro**	**;define a macro that places the ;length of a string in a byte in ;front of the string ltext: macro dc.b \@2-\@1 \@1: dc.b \1 ; text \@2: endm**
mexit	**terminate a macro definition**	**mexit**
mlist [on|off]	**turn on or off listing of macro expansion**	**mlist on**
nolist	**turn off listing**	**nolist**
nopage	**disable pagination in the listing file**	**nopage**
offset <expression>	**start an absolute section that will be used to define symbols and not to produce any code or data. The section starts at the address specified in <expression> and remains active while no directive or instruction producing code or data is entered.**	**offset 0 next: ds.b 2 buffer: ds.b 80 size: ldy next,x ;end of ;offset ;section**
org <expression>	**set the location counter to an offset from the beginning of a program section. <expression> must be a valid absolute expression and must not contain any forward or external references.**	**org $e000**
page	**start a new page in the listing file**	**page**
plen <page_length>	**specify the number of lines per page in the listing file**	**plen 58**
<section_name>:section [<attributes>]	**define a new program section; attribute keywords are: abs for absolute section, bss for bss section (no data), hilo for values to be stored in descending order or significance, even for enforce even starting address and size, zpage to enforce 8-bit relocation. When -m is specified, this directive also accepts a number as the operand.**	

label: set <expression>	**give a value to a symbol which can be reset**	**OFST: set 10**
spc <num_lines>	**insert a number of blank lines before the next statement in the listing file.**	**spc 5** **title "new file"**
switch <section_name>	**switches output to the section defined with the section directive named <section_name>**	**switch .bss** **buffer: ds.b 512** **xdef buffer**
tabs <tab_size>	**specify the number of spaces for a tab character in a listing file**	**tabs 6**
title "name"	**set default page header**	**title "My application"**
xdef identifier[,identifier…]	**declare a variable to be visible. Asymbol may only be declared as xdef in one module.**	**xdef sqrt ; allow sqrt to be** **; called from another** **; module**
xref[.b] identifier[,identifier…]	**declare a variable as being defined elsewhere; symbols defined in other modules must be declared as xref. A symbol may be declared both xdef and xref in the same module, to allow for usage in common header files. The directive xref.b declares external symbols located in the .bsct section.**	**xref otherprog** **xref.b zpage**

Linker Considerations

The V4.1x C compiler package now includes linker support for **static stack analysis**. If you run the linker with the *-m file.map* option to generate a linker map file, the linker will include an analysis of stack usage in all the functions that make up your application. This is a useful feature as it gives a rough estimate of the size of the stack you will need to run the application.

The stack analysis includes the function name, a '>' marker if the function is not called by any other function (e.g. main() or functions marked with @interrupt), followed by the total number of bytes needed on the stack when the function is called. The next number, in braces, shows the size of the locals on the stack for the function, not including any internal calls which are generated by the compiler, so if there are no internal calls the two figures match. This last field is replaced by a *** recursive *** message if the function is recursive.

Example Linker Command Files

The linker is your primary tool to place code and data in pre-defined, absolute locations in the target microcontroller's memory.

68HC812A4 Target

As an example, let's say the target is a Motorola MC68HC12A4 running in expanded mode with the following configuration:

On-chip RAM sixe:	1024 bytes
On-chip RAM address:	0x800 to 0xBFF
On-chip EEPROM size:	4096 bytes
On-chip EEPROM address:	0x1000 to 0x1FFF
Off-chip ROM size:	32 Kbytes
Off-chip ROM address:	0x2000
Register block size:	512 bytes
Register block address:	0x000 to 0x1FF

Interrupt vectors size: 64 bytes
Interrupt vectors address: 0xFFC0 to 0xFFFF

To link your application code, *test.o*, for this configuration, the linker reads a linker command file, *test.lkf*, which tells it where to place target code and data. Below is an example *.lkf* file for a MC68HC12A4 in expanded mode:

```
#         link command file for MC68HC12A4
#         Copyright (c) 1995 by COSMIC Software
#
+seg .text -b 0x2000 -m32768 -n .ROM  # 32kb ROM start address and size (.text section is program code.
+seg .const -a .ROM                   # constants follow .ROM section
                                      # .ROM and .const sections are located at target ROM
+seg .data -b 0x800  -m768 -n .IRAM   # RAM start address and maximum allowable size for initialized
                                      # data. Allow 256 bytes for stack space (768+256 = 1024)
+seg .bss -a .IRAM                    # uninitialized static data
+seg .eeprom -b 0x1000 -m4096         # EEPROM start address and size
crts.o                                # C startup routine
test.o                                # application program
c:/c/6812/lib/libi.h12                # Integer C library (if needed)
c:/c/6812/lib/libm.h12                # machine support library
+seg .const -b 0xffc0                 # interrupt vectors start address
vector.o                              # pre-compiled interrupt vectors file
+def __memory=@.bss                   # symbol used by library
+def __stack=0xBFF                    # stack pointer initial value
```

In the above example, the line:

```
+seg .text -b 0x2000 -m32768 -n .ROM  # 32kb ROM start address and size (.text section is program
                                      # code). Name this .text section .ROM
```

places the .text sections of the following files, *crts.o* and *test.o* starting at address 0x2000 (-b 0x2000), which is the starting address of target system ROM, into a program section named .ROM and checks that the total size of this section does not exceed 32768 bytes (-m32768) – if it does the linker will generate an overflow message; the part -n .ROM tells the linker to give this section the name .ROM, which will be used in the next line to place the .const program section.

```
+seg .const -a .ROM                   # .const section (constants) follow section called .ROM
```

The above line tells the linker to place the .const program sections from the files, *crts.o* and *test.o*, after the last address allocated to the .text section named .ROM. If your application code contains string constants, such as strings inside a printf() statement, or another kind of literal, these will be allocated into the .const program section; if your application does not contain any constants, the .const section will be empty, unless you have linked in run-time library routines that contain constants. Note, that because the .const section follows the end of the .ROM section (-a .ROM), the linker will check the combined size of the .ROM and .const sections, and will diagnose an overflow error if the combined sizes exceed the 32768 limit. If the .const section is biased separately (e.g. +seg .const -b 0x8000 -m 512), then the .ROM and .const sections are no longer considered as one and the linker will check for overflow in each section independently. Note also that you can compile your application code with the +nocst option which forces constants into the .text section so that there is no .const section.

68HC912B32 Target

Now let's look at the configuration of a 68HC912B32 running in single-chip mode:

On-chip RAM size:	1024 bytes
On-chip RAM address:	0x800 to 0xBFF
On-chip FLASH EEPROM size:	32 Kbytes
On-chip FLASH EEPROM address:	0x8000
Register block size:	512 bytes
Register block address:	0x000 to 0x1FF
Interrupt vectors size:	64 bytes
Interrupt vectors address:	0xFFC0 to 0xFFFF

The link command file below is suitable for a MC68HC912B32 target in single-chip mode:

```
+seg .text -b 0x8000 -m32768 -n .ROM  # 32kb ROM start address and size (.text section is program code.
+seg .const -a .ROM                   # constants follow .ROM section
                                      # .ROM and .const sections are located at target ROM
+seg .data -b 0x800  -m768 -n .IRAM   # RAM start address and maximum allowable size for initialized
                                      # data. Allow 256 bytes for stack space (768+256 = 1024)
+seg .bss -a .IRAM                    # uninitialized static data
crts.o                                # C startup routine
test.o                                # application program
c:/c/6812/lib/libi.h12                # Integer C library (if needed)
c:/c/6812/lib/libm.h12                # machine support library
+seg .const -b 0xffc0                 # interrupt vectors start address
vector.o                              # pre-compiled interrupt vectors file
+def __memory=@.bss                   # symbol used by library
+def __stack=0xBFF                    # stack pointer initial value
```

Support for P&E Debugger

If you are using the P&E assembler, debugger and BDM cable, COSMIC can supply a converter program, ***cvpne***, to convert the *.h12* file generated by the linker into a .MAP format file which can be read by the P&E debugger. If you need this converter, make a support request, via email, to ***support@cosmic-us.com*** or go to the Support page on our web-site at ***www.cosmic-us.com*** or call + ***781 952-2556*** and ask for support.

Here is a brief summary of the CVPNE utility:

Description

Generate P&E mapfile format

Syntax

cvpne -[c o* u] <file>

Function

cvpne is the utility used to generate the P&E mapfile format from a relocatable (.o) or executable (.h12) file. cvpne accepts the following options:

-c - This flag is reserved for future use (it sets the most significant bit of line numbers to 1; this may be used in a future release of the P&E software).

-o* - where * is a filename. * is used to specify an output file for cvpne. By default, if -o is not

specified, cvpne sends its output to a filename which is built from the input filename by replacing the filename extension by ".map".

-u - DO NOT convert labels to upper case in the output file. By default, cvpne outputs symbols and filenames in uppercase.

Example

```
cvpne acia.h12
```

generates an output file called acia.map

Libraries Considerations

The sizes (in bytes) of the V4.1x 68HC12 C compiler machine library (libm.h12 in the Lib directory) functions are given in the table below. These library routines are called directly by the compiler to assist with operations (e.g. floating point or long integer arithmetic, EEPROM support) which are too lengthy to generate in-line code.

bfget.o – 50 bfput.o – 64 check.o – 1 dadd.o – 500 dcmp.o – 53 ddiv.o – 204 dmul.o – 365 dneg.o – 11 dtod.o – 17 dtof.o – 90 dtol.o – 138 dtos.o -- 19 eepbfb.o – 26 eepbfd.o – 41 eepbfx.o – 63 eepdbl.o – 41 eeprom.o – 165 eepstr.o -- 47	fadd.o – 263 fcmp.o – 39 fdiv.o – 124 fgadd.o – 7 fgdiv.o – 7 fgmul.o – 7 fgsub.o – 7 fmul.o – 132 ftod.o – 58 ftol.o – 105 itod.o – 58 itof.o – 49 jltab.o – 68 jtab.o – 41 ladd.o – 11 land.o – 13 lcmp.o - 19 ldiv.o – 208	lgadd.o – 13 lgand.o – 15 lgdiv.o – 28 lglsh.o – 36 lgmul.o – 32 lgop.o – 22 lgor.o – 15 lgrsh.o -- 40 lgsub.o – 18 lgursh.o – 36 lgxor.o – 15 llsh.o – 14 lmul.o – 28 lneg.o – 13 lor.o – 13 lrsh.o – 15 lrzmp.o – 11 lsub.o – 11	ltod.o – 85 ltof.o – 77 lursh.o – 14 lxor.o – 13 lzmp.o – 13 uitod.o – 46 uitof.o – 37 ultod.o – 72 ultof.o -- 59
1895 bytes	1216 bytes	375 bytes	416 bytes
TOTAL SIZE = 3902 BYTES			

The COSMIC linker only loads those library functions which it needs to complete a user application link, but some users want to put the whole run-time library in a separate ROM, which is not changed even as the application changes. The ***clib*** librarian allows you to force load an entire library at link time, so you should remove machine library routines that you know your application will never need, thus saving ROM space.

Over 80% of the 68HC12 machine library functions provide support for:

(1) double precision (64-bit) floating point math (the routines beginning with “d” – total size of 1573 bytes – remove them and the library is now 3902 - 1573 = **2329** bytes),
(2) single precision (32-bit) floating point math (the routines beginning with “f” – total size of 971 bytes – remove them and the library is now 2329 - 971 = **1358** bytes)
(3) long integer (32-bit int) math (the routines beginning with “l” – total size of 667 bytes – remove them and the library is now 1358 - 667 = **691** bytes)

If you are not using a version of the chip with EEPROM support or you are not using the @eeprom keyword in your C source code, you can remove the EEPROM support routines (beginning with “ee” – total size 383 bytes – remove them and the library is now 691 - 383 = **308** bytes).

Run-time libraries are provided in pre-built binary form and in source form and can be freely modified to meet your needs.

IEEE695 Object File Format Support

The COSMIC C compiler package can be used to generate IEEE695 object file format that is a common format read by third-party C source-level debuggers e.g. Nohau or Noral. The ***cv695*** utility converts the *.h12* output file generated by the COSMIC linker into an IEEE695 format file. The manual page below describes its use:

Programming Utility cv695

NAME

cv695 - generate IEEE695 format

SYNOPSIS

cv695 -[d mod? o* v] <file>

FUNCTION

cv695 is the utility used to convert a file produced by the linker into an IEEE695 format file.

The flags to cv695 are:

+V4 - This option is used only for older IEEE loaders. i.e. loaders designed to work with version 4.X of this utility.

-d - dumps to the screen information such as: frame coding, register coding, e.g. all the processor specific coding for IEEE (note: some of these codings have been chosen by COSMIC because no specifications exists for them in the current published standard).

THIS INFORMATION IS ONLY RELEVANT FOR WRITING AN IEEE695 FORMAT READER.

-mod? where ? is a character used to specify the compilation model selected for the file to be converted.

THIS FLAG IS CURRENTLY ONLY MEANINGFUL FOR 68HC16 TARGET

-o* where * is a filename. * is used to specify the output file for cv695. By default, if -o is not specified, cv695 sends its output to the file whose name is obtained from the input file by replacing the filename extension with ".695".

-v selects verbose mode. cv695 will display information about its activity.

EXAMPLES: Under MS/DOS, the command could be:

```
C>cv695 -modt C:\test\basic.h16
```

and will produce C:\test\basic.695 and the following command:

```
C>cv695 -o file basic.h12
```

will produce an IEEE695 file named *file*

Under UNIX, the command could be:

```
% cv695 /test/basic.h12
```

and will produce test/basic.695

Disclaimer

COSMIC Software provides this document “as-is” and does not guarantee that it is free from errors. – feel free to use it as an aid to understanding. Any inconsistencies between this document and the V4.x 68HC12 C compiler product release you are using should be reported to COSMIC at + (781) 932-2556 or email to sales@cosmic-us.com.