COSMIC C Cross Compiler For Motorola CPU12 Family

COSMIC's C cross compiler, ***cx6812*** *for the Motorola 68HC12 and* ***MCS12*** *families of microcontrollers, incorporates over twenty years of innovative design and development effort. In the field since 1994,* ***cx6812*** *is reliable and field-tested, and incorporates many features to help* ***ensure your embedded CPU12 design meets and exceeds performance specifications****.*

The **C Compiler** package for Windows includes: COSMIC integrated development environment (IDEA), optimizing C cross compiler, macro assembler, linker, librarian, object inspector, hex file generator, object format converters, debugging support utilities, run-time libraries and a compiler command driver. The PC compiler package runs under Windows 95/98/ME/NT4/2000 and XP.

Key Features

Optimized to 68HC12 and MCS12 Instructions and Addressing Modes,
Global and Processor-Specific Optimizations,
Optimized Function Calling,
Function In-Lining,
ANSI C Implementation,
Automatic Checksums,
Built-In Fuzzy Logic Support,
C support for Internal EEPROM,
C support for Direct Page Data,
C support for Code Bank Switching,
C support for Interrupt Handlers,
Three In-Line Assembly Methods,
Extensions to ANSI for Embedded Systems,
User-defined Code/Data Program Sections,
Single and Double Precision Float Support,
Motorola MCUasm™ Compatible Assembler,
Assembler Supports C #defines,
Absolute C and Assembly Listings,
Static Analysis of Stack Usage,
Royalty-Free Library Source Code,
Works With All Popular 68HC12 In-Circuit Emulators,
IEEE-695, ELF/DWARF and P&E Debug support
First Year of Support Service Included with No Charge Upgrades.

Microcontroller-Specific Design

cx6812, is designed specifically for the Motorola 68HC12 and MCS12 families of microcontrollers; all HC12/MCS12 processors are supported. A special processor specific code generator and optimizer eliminates the overhead and complexity of a more generic compiler. You also receive header file support for many of the popular HC12/MCS12 peripherals, so you can access their memory mapped objects by name either at the C or assembly language levels.

ANSI C

This implementation conforms with the **ANSI** and **ISO Standard C** specifications which helps you protect your software investment by aiding code portability and reliability.

Flexible User Interface

The Cosmic C compiler can be used with the included Windows IDEA or as a Windows 32-bit command line application for use with your favorite editor, make or source code control system. It's your choice!!

Automatic Checksum

The linker can automatically create and maintain a multiple segment check sum mechanism in your application. Choose either an 8 or 16-bit checksum algorithm. Call one of the included checksum verify functions from any part of your application to calculate and compare the checksums.

Function Copy (Boot loader)

Create a block of functions that is stored in ROM and copied to RAM by an included library routine. Multiple blocks may be setup and copied and/or executed independently. Separate copies of library routines may also be included in the RAM code without symbol conflicts.

C Runtime Support

C runtime support consists of a subset of the standard ANSI library, and is provided in C source form with the binary package so you are free to modify library routines to match your needs. The basic library set includes the support functions required by a typical embedded system application. All runtime library functions are **ROMable** and reentrant. Support includes:

- Character handling
- Mathematical functions
- Non-local jumps
- Formatted serial input/output
- String handling
- Memory management

The package provides both an **integer-only library** as well as the standard **double and single precision floating point library**. This allows you to select the smaller and faster integer-only functions, if your application does not require floating point support.

Optimizations

cx6812 employs global and microcontroller-specific optimizations which help ensure your embedded application will **meet and exceed its performance specifications**. You retain control over optimizations via compile-time options and keyword extensions to ANSI C, so you can fine tune your application code to match your design specification:

- fully optimized code can be debugged, without change, using COSMIC's **ZAP/SDI12** and **ZAP/SIM12** debuggers or third-party debuggers that read IEEE695 format files,
- ***cx6812*** supports **global optimizations** which allow it to optimize whole C functions as well as C statements,
- ***cx6812***'s **peephole optimizer** adds further optimization by replacing inefficient code sequences with optimal code sequences for the 68HC12,
- Functions which require a stack can be marked, using the **@fast** keyword, as requiring fast entry/exit code sequences in which case ***cx6812*** will generate entry/exit code in-line rather than by calling a machine library,
- Selected string library functions can be in-lined,
- ***cx6812*** can reorder function local data so that the most referenced locals are allocated stack space closest to the function frame pointer for fast access,
- Function arguments are passed in registers when possible, and *char*-sized data can be passed without widening to *int*,
- Commonly used static data can be selectively, using the **@dir** keyword, or globally, using a compile-time option, placed into direct page memory (the first 256 bytes of memory) to decrease access time,
- ***cx6812*** makes full use of the IX and IY index registers for addressing and pointer operations,
- The 68HC12 bit instructions (bclr,bset,brclr,brset) are used extensively for bit operations,
- Arithmetic operations are performed in *char* precision if the types are 8-bit,
- Strict single-precision (32-bit) or strict double-precision (64-bit) floating point arithmetic and math functions. Floating point numbers are represented as in the IEEE 754 Floating Point Standard,
- Other optimizations include: branch shortening logic, jump-to-jump elimination, constant folding, elimination of unreachable code, removal of redundant loads/stores, and switch statement optimizations.

Extensions to ANSI C

cx6812 includes several extensions to the ANSI C standard which have been designed specifically to give you **maximum control** of your application at the C level and to **simplify** the job of writing C code for your embedded 68HC12 design:

- The **@far** keyword can be attached to a C function declaration to instruct the compiler to generate the 68HC12 *call/ rtc* instruction pair to access function code in expanded memory,
- The **@eeprom** modifier can be attached to a C data declaration to inform the compiler that the data object resides in 68HC12 EEPROM space; the compiler will automatically generate the required code sequence when writing to the EEPROM location,
- The **_asm()** statement can be used to insert assembly instructions directly in your C code to avoid the overhead of calling assembly language functions. **_asm()** statements can only be used within C function code and can be used in C expressions,
- Arguments can be passed into **_asm()** assembly language statements to allow access to C local variables from assembly language code,
- Assembly language statements can be inserted inside or outside of C functions using **#pragma asm .. #pragma endasm** or the alias **#asm .. #endasm**,
- User-defined program sections are supported at the C and assembler levels: **#pragma section <*name*>** declares a new program section *name* for code or data which can be located separately at link time,
- The **@interrupt** keyword can be attached to a C function definition to declare the function as an interrupt service

routine. The compiler preserves volatile registers not already saved by the processor,

- **@<address>** syntax allows an absolute address to be attached to a data or function definition; this is useful for interrupt handlers written in C and for defining memory mapped I/O,
- *char (8-bit), int (16-bit)* or *long int (32-bit)* bitfields can be defined, and bit-numbering from right-to-left or left-to-right can be selected,
- C functions returning Boolean 1 or 0 can be defined as **@bool** functions to optimize function return code.

Fuzzy Logic Support

cx6812 provides a set of library functions which support direct access from C to specific 68HC12 fuzzy logic instructions: *memhc12()* to fuzzify input variables using **mem**, *revhc12()* to evaluate rules using **rev**, *revwhc12()* to evaluate rules using **revw** and *wavhc12()* to defuzzify outputs using the **wav** and **ediv** instructions.

Additional Compiler Features

- **Full C source-level debugging support**. There is no limit on the size of the debug section,
- Absolute and relocatable listing file output, with interspersed C, assembly language and object code; optionally, you can include compiler errors and compiler optimization comments,
- Extensive and useful compile-time error diagnostics,
- Fast compile and assemble time,
- Full user control over include file path(s), and placement of output object, listing and error file(s),
- All objects are relocatable and host independent. (i.e. files can be compiled on a workstation and debugged on a PC),
- Function code and switch tables are generated into the code (*.text*) section. Constant data such as string constants and *const* data are generated into a separate *.const* program section,
- Initialized static data can be located separately from uninitialized data,
- All function code is (by default) reentrant, never self-modifying, including structure assignment and function calls, so it can be shared and placed in ROM,
- Code is generated as a symbolic assembly language file so you can examine compiler output,
- ***cx6812*** creates all its tables dynamically on the heap, allowing you to **compile large source files**,
- Unused variables can be flagged with an error message.
- Common string manipulation routines are implemented in assembly language for fast execution.

68HC12 Assembler

The COSMIC 68HC12 assembler, ***ca6812***, **conforms to the standard Motorola syntax** as described in the document *Assembly Language Input Standard*; ***ca6812*** supports macros, conditional assembly, includes, branch optimizations, expression evaluation, relocatable or absolute output, relocatable arithmetic, listing files and cross references. **Assembler accepts C syntax for #includes and #defines** so include files can be shared between C and Assembly modules. The assembler also creates full debug information, so that debuggers can perform full source-level debug at the assembly language level.

Linker

The COSMIC linker, ***clnk***, combines relocatable object files created by the assembler, selectively loading from libraries of object files made with the librarian, ***clib***, to create an executable format file. ***clnk*** features:

- Flexible and extensive user-control over the linking process and selective placement of user application code and data program sections,
- ***clnk*** analyzes stack usage for local variables and function arguments and the function calling hierarchy to provide a **static analysis of stack usage** which helps you understand how much stack space your application needs,
- Multi-segment image construction, with user control over the address for each code and data section. Specified addresses can cover the full logical address space of the target processor with up to 255 separate segments. This feature is useful for creating an image which resides in a target memory configuration consisting of scattered areas of ROM and RAM,
- Generation of memory map information to assist debugging,
- All symbols and relocation items can be made absolute to prelocate code that will be linked in elsewhere,
- Symbols can be defined, or aliased, from the Linker command File.

Librarian

The COSMIC librarian, ***clib***, is a development aid which allows you to collect related files into one named library file, for more convenient storage. ***clib*** provides the functions necessary to build and maintain object module libraries. The most obvious use for ***clib*** is to collect related object files into separate named library files, for scanning by the linker. The linker loads from a library only those modules needed to satisfy outstanding references.

Object Module Inspector

The COSMIC object module inspector, ***cobj***, allows you to examine library and relocatable object files for symbol table and file information. This information is an essential aid to program debugging.

- Symbol table cross referencing,
- Section sizes of the individual program sections can be printed for object and library files,
- Program segment map: lists all program segments, their sizes, absolute addresses and offsets.

Absolute Hex File Generator

The COSMIC hex file generator, ***chex***, translates executable images produced by the linker to one of several hexadecimal interchange formats for use with most common In-Circuit Emulators and PROM programmers:

- Standard **Intel hex** format,
- **Motorola S-record** and S2 record format,
- Rebiasing of text and data section load addresses.

Bank Packing Utility

For users of code bank-switching, the ***cbank*** utility program takes a user-specified list of object files, and creates an output file, which can be input to the clnk linker, that contains an optimized ordered list of object files so as to minimize memory holes at bank boundaries. This utility saves the user from having to figure out which modules can fit into which bank.

Absolute C and Assembly Listings

Paginated listings can be produced to assist program understanding. Listings can include original C source code with interspersed assembly code and absolute object code. Optionally, you can include compiler errors and optimization comments.

Third Party Debugging Support

You can use ***cx6812*** and ***ca6812*** with **ZAP SIM, ZAP CABLE12(HS).** **CX*6812*** also supports several standard debugging formats for use with third party debuggers and logic analyzers. Supported Debug formats include **IEEE-695, ELF/DWARF** and P&E map file format. Third party vendors include logic analyzers from HP and Tektronix and in-circuit emulators from Isystem, Lauterbach and NOHAU.

Packaging

All compiler packages are provided on standard CD-ROM with complete on-line user documentation in Adobe PDF format.

The **C Compiler** package for Windows includes: An integrated development environment (IDEA), optimizing C cross compiler, macro assembler, linker, librarian, object inspector, hex file generator, object format converters, debugging support utilities, run-time libraries and a compiler command driver. The PC compiler package runs under Windows 95/98/ME and Windows NT4/2000/XP.

The Windows version also includes integration files for Starbase's popular **CodeWright®** Windows® code and project editor and GNU make utility.

The **C Compiler** package for UNIX includes: *An* optimizing C cross compiler, macro assembler, linker, librarian, object inspector, hex file generator, object format converters, debugging support utilities, run-time libraries and a compiler command driver. The UNIX compiler package is available for SUN Solaris, HP-UX and PC Linux.

Support Services

All COSMIC Software products come with the first year of support included in the price. You will receive a courteous and prompt service from our technical support staff and **you retain control of the severity of the problem** i.e. if it's a problem that is critical to your project we guarantee you a response time of one to three business days depending on the severity of the problem. Service is provided during normal business hours EST via email, fax or telephone and is **unlimited** while you have a valid annual support agreement. New releases of the software are provided **free of charge** to customers with a current service agreement.

Ordering Information

cx6812 package product codes are as follows:

Host System	Product Code
PC MS Windows Windows 95/98/ME/NT4/2000/XP-	CWSH12
PC Linux	CLXH12
SUN SPARC(SunOS/Solaris)	CSSH12
HP9000(HPUX)	CHPH12

Orders are shipped within one week of receipt of hard copy purchase order. Call our sales department for license fees and multiple copy discounts.

Other COSMIC Software Products

COSMIC Software products focus on Motorola 8,16 and 32-bit microcontrollers. C compiler/debugger support is available for **68HC05, 68HC08, 6809, 68HC11, 68HC12, 68HC16, 683XX and 680X0**. For more information on the ZAP C and assembler source-level debugger, ask for the ZAP Product Description and demo CD.

Supporting Embedded Innovation since 1983

Digital DNA from Motorola DEVELOPER

For Sales Information please contact:

COSMIC Software USA
COSMIC Software, Inc.
400 West Cummings Park, Suite 6000
Woburn, MA 01801-6512 USA
Phone: (781) 932-2556 Fax: (781) 932-2557
Email: sales@cosmic-us.com
web: www.cosmic-software.com

COSMIC Software France
33 Rue Le Corbusier, Europarc Creteil
94035 Creteil Cedex France
Phone: + 33 4399 5390 Fax: + 33 4399 1483
Email: sales@cosmic.cosmic.fr
web: www.cosmic.fr

COSMIC Software UK
Oakwood House
Wield Road, Medstead
Alton, Hampshire
GU34 5NJ, U.K.
Phone: +44 (0)1420 563498 Fax: +44 (0)1420 561946
Email: *sales@cosmic.co.uk*

COSMIC Software GmbH
Rohrackerstr 68 D-70329 Stuttgart Germany
Tel.+ 49 (0)711 4204062 Fax + 49 (0)711 4204068
Email: sales@cosmic-software.de
web: www.cosmic-software.de

COSMIC MC68HC12 and MC68HC11 C Compiler Comparison

cx6812 is not a generic 68HC11 compiler that generates 68HC11 code to run on a 68HC12. ***cx6812*** is tuned to take full advantage of the additional 68HC12 instructions and addressing modes to generate optimal code.

Example 1.

Operations on dynamic (local or stack) data are considerably simplified for the 68HC12 because the stack is directly addressable:

68HC11:

```
tsx         ;load stack address in X  (1 byte,3 cycles)
ldd 2,x     ;load and                 (2 bytes, 5 cycles)
std 4,x     ;store variable           (2 bytes, 5 cycles)
```

and computing the address of a dynamic variable needs more instructions:

```
tsx         ;load stack address       (1 byte, 3 cycles)
xgdx            ;in the D register        (1 byte, 3 cycles)
addd    #4      ;add offset               (3 bytes, 4 cycles)
```

68HC12:

```
ldd 4,sp    ;load and                 (1 byte, 3 cycles)
std 4,sp    ;store a variable         (1 byte, 3 cycles)
```

and a stack address is easily computed:

```
leax    4,sp    ;load effective address    (2 bytes, 2 cycles)
```

Example 2.

Allocation of a function stack frame to host dynamic data and initialize a “frame pointer” to access the data is also more efficient on the 68HC12:

68HC11:

```
tsx         ;load stack address       (1 byte, 3 cycles)
xgdx            ;in D register            (1 byte, 3 cycles)
subd    #size   ;compute the new      (3 bytes, 4 cycles)
xgdx            ;stack pointer            (1 byte, 3 cycles)
txs         ;and set it               (1 byte, 3 cycles)
```

The same sequence is used at the end of the function to deallocate the stack frame.

68HC12:

```
leas    -size,sp    ;open stack frame(2-4 bytes, 2 cycles)
```

Only one instruction is needed to open and close the frame.

Because the stack is directly addressable on the 68HC12, the compiler does not need a dedicated frame pointer, which frees up another register for code generation; it also saves a word of storage on the stack, as the compiler no longer has to save the frame pointer at the entry and exit of each C function.

Example 3:

C pointers are implemented in index registers X and Y as these are the only registers that allow memory indirection. Register D can still be used when no memory indirection is required, for pointer assignment for example. Notice that X and Y is equivalent on the 68HC12 but on the 68HC11 Y costs one extra byte and 1 extra cycle.

Array indexing:
i = tab[j];

68HC11:

```
ldd     j       ; (2-3 bytes, 4-5 cycles)
addd    #tab    ; (3 bytes, 4 cycles)
xgdx            ; (1 byte, 3 cycles)
ldd     0,x     ; (2 bytes, 5 cycles)
std     i       ; (2-3 bytes, 4-5 cycles)
```

68HC12:

```
ldx     j       ; (2-3 bytes, 3 cycles)
ldd     tab,x   ; (2-4 bytes, 3-4 cycles)
std     i       ; (2-3 bytes, 2-3 cycles)
```

Note: the above 68HC12 code sequence can also apply to the 68HC11 if the array is located in the direct page, usually the first 256 bytes of memory.

Pointer indexing:
i = p[j];

68HC11:

```
ldd     j       ; (2-3 bytes, 4-5 cycles)
addd    p       ; (2-3 bytes, 5-6 cycles)
xgdx            ; (1 byte, 3 cycles)
ldd     0,x     ; (2 bytes, 5 cycles)
std     i       ; (2-3 bytes, 4-5 cycles)
```

68HC12:

```
ldx     p       ; (2-3 bytes, 3 cycles)
ldd     j       ; (2-3 bytes, 3 cycles)
ldd     d,x     ; (2 bytes, 3 cycles)
std     i       ;2-3 bytes, 4-5 cycles)
```

Auto-increment:
i = *p++;

68HC11:
```
ldx   p      ; (2-3 bytes, 4-5 cycles)
ldd   0,x    ; (2 bytes, 5 cycles)
inx          ; (1 byte, 3 cycles)
inx          ; (1 byte, 3 cycles)
stx   p      ; (2-3 bytes, 4-5 cycles)
std   i      ; (2-3 bytes, 4-5 cycles)
```

68HC12:
```
ldx   p      ; (2-3 bytes, 4-5 cycles)
ldd   2,x+   ; (2 bytes, 3 cycles)
stx   p      ; (2-3 bytes, 2-3 cycles)
std   i      ; (2-3 bytes, 2-3 cycles)
```

Access to an array of pointers with an index:
i = *ptab[i];

68HC11:
```
ldd   i      ; (2-3 bytes, 4-5 cycles)
lsld         ; (1 byte, 3 cycles)
addd  #ptab  ; (3 bytes, 4 cycles)
xgdx         ; (1 byte, 3 cycles)
ldx   0,x    ; (2 bytes, 5 cycles)
ldd   0,x    ; (2 bytes, 5 cycles)
std   i      ; (2-3 bytes, 4-5 cycles)
```

68HC12:
```
ldx   #ptab ; (3 bytes, 2 cycles)
ldd   i     ; (2-3 bytes, 3 cycles)
lsld        ; (1 byte, 1 cycle)
ldd   [d,x] ; (2 bytes, 6 cycles)
std   i     ; (2-3 bytes, 2-3 cycles)
```

The above 68HC11 examples use the X register; if the Y register is used the code will be larger and slower.

Example 4:

C structures can be copied into another structure variable, or copied onto the stack to be passed as a function argument. The 68HC12 offers efficient ‘move’ instructions which can be used along with the loop instructions. Assuming that source and destination addresses are loaded into X and Y:

68HC11:
```
ldaa  #size      ; (2 bytes, 2 cycles)
loop:
ldab  0,x        ; (2 bytes, 4 cycles)
stab  0,y        ; (2 bytes, 4 cycles)
inx              ; (1 byte, 3 cycles)
iny              ; (2 bytes, 4 cycles)
deca             ; (1 byte, 2 cycles)
bne   loop       ; (2 bytes, 3 cycles)
```

68HC12:
```
ldd   #size      ; (3 bytes, 2 cycles)
loop:
movb  1,x+,1,y+  ; (4 bytes, 5 cycles)
dbne  d,loop     ; (3 bytes, 3 cycles)
```

or even better to copy using ‘move word’:

```
ldd   #size/2; (3 bytes, 2 cycles)
loop:
movw  2,x+,2,y+  ; (4 bytes, 5 cycles)
dbne  d,loop     ; (3 bytes, 3 cycles)
movb  0,x,0,y    ; if size is odd (4 bytes, 5 cycles)
```

Example 5:

In complex expressions, intermediate results need to be stored in memory or another register, for instance:

```
a = (b & c) | (d & e);      /* all char's */
```

68HC11:
```
ldab  b          ; (2-3 bytes, 3-4 cycles)
andb  c          ; (2-3 bytes, 3-4 cycles)
pshb             ; (1 byte, 3 cycles)
ldab  d          ; (2-3 bytes, 3-4 cycles)
stab  d          ; (2-3 bytes, 3-4 cycles)
tsx              ; (1 byte, 3 cycles)
orab  0,x        ; (2 bytes, 4 cycles)
ins              ; (1 byte, 3 cycles)
stab  a          ; (2-3 bytes, 3-4 cycles)
```

68HC12:
```
ldab    b           ; (2-3 bytes, 3 cycles)
andb    c           ; (2-3 bytes, 3 cycles)
pshb                ; (1 byte, 2 cycles)
ldab    d           ; (2-3 bytes, 3 cycles)
andb    e           ; (2-3 bytes, 3 cycles)
orab    1,sp+       ; (2 bytes, 3 cycles)
stab    a           ; (2-3 bytes, 2 cycles)
```

Example 6:

The 68HC12 provides four multiply instructions:

```
mul     ;unsigned 8 x 8 -> 16, same as 68HC11
emul    ;unsigned 16 x 16 -> 32
emuls   ;signed 16 x 16 -> 32
emacs   ;signed 16 x 16 -> 32 with accumulate in mem
```

The ‘emul’ instruction can be used to implement the 16-bit multiplication assuming all variables are ‘int’ types:

```
a = b * c;
```

68HC12:
```
ldd     b           ; (2-3 bytes, 3 cycles)
ldy     c           ; (2-3 bytes, 3 cycles)
emul                    ; (1 byte, 3 cycles)
std     a           ; (2-3 bytes, 2-3 cycles)
```

To get a full 32-bit result, the proper C code is:

```
la = (long) b *  c;
```

which for the 68HC12 generates:

```
ldd     b           ; (2-3 bytes, 3 cycles)
ldy     c           ; (2-3 bytes, 3 cycles)
emuls               ; (1 byte, 3 cycles)
std     a+2         ; (2-3 bytes, 2-3 cycles)
sty     a           ; (2-3 bytes, 2-3 cycles)
```

Example 7:

The 68HC12 provides five divide instructions:

```
idiv            ;unsigned 16/16, same as 68HC11
fdiv            ;fractional unsigned 16/16, same as 6811
idivs           ;signed 16 / 16
ediv            ;unsigned 32/16
edivs           ;signed 32/16
```

```
a = b / c;
```

68HC12:
```
ldd     b       ; (2-3 bytes, 3 cycles)
ldx     c       ; (2-3 bytes, 3 cycles)
idivs               ; (2 bytes, 12 cycles)
stx     a       ; (2-3 bytes, 2-3 cycles)
```

The ‘idiv’ instruction is used when operands are unsigned.

```
a = (int) (la / b);
```

68HC12:
```
ldd     la+2    ; (2-3 bytes, 3 cycles)
ldy     la      ; (2-3 bytes, 3 cycles)
ldx     b       ; (2-3 bytes, 3 cycles)
edivs               ; (2 bytes, 12 cycles)
sty     a       ; (2-3 bytes, 2-3 cycles)
```

```
a = max (b, c);
```

68HC12:
```
ldd     2,sp        ; (2 bytes, 3 cycles)
emaxd   4,sp        ; (3 bytes, 4 cycles)
std     6,sp        ; (2 bytes, 2 cycles)
```

Example 8:

The 68HC12 provides relative conditional branches with 16-bit offsets, allowing all of memory to be addressed; it also provides a set of special branches:

```
if (a == 1)
    …
```

68HC12:
```
ldd     a       ; (2-3 bytes, 3 cycles)
dbne    d,endif; (3 bytes, 3 cycles)
…
endif:
```

```
if (func() != 0)
```

68HC12:
```
jsr     func    ; result in D (2-3 bytes, 4 cycles)
tbeq    d,endif; (3 bytes, 3 cycles)
```

```
a = b << c;
```

68HC12:

```
ldd     b       ; (2-3 bytes, 3 cycles)
ldx     c       ; (2-3 bytes, 3 cycles)
beq     nosh    ; (2 bytes, 3 cycles)
loop:
lsld            ; (1 byte, 1 cycle)
dbne    x,loop  ; (3 bytes, 3 cycles)
nosh:
```

Small lists of consecutive values in a C ‘switch’ statement can be implemented using the decrement and branch instructions:

```
switch(i)
    {
case 0:
…
```

68HC12:

```
ldd     i           ; (2-3 bytes, 3 cycles)
beq     case_0      ; (2 bytes, 1 or 3 cycles)
dbne    d,case_1    ; (3 bytes, 3 cycles)
dbne    d,case_2    ; (3 bytes, 3 cycles)
```

The 68HC12 PC-relative indexed indirect address mode can be used for longer contiguous lists.