Freescale Semiconductor
Application Note

AN2111
Rev. 2, 11/2004

A Coding Standard for HCS08 Assembly Language

by: Jim Sibigtroth
Applications and Systems Engineering
Austin, Texas

Introduction

This application note details an HCS08 assembly language coding standard. Freescale's 8/16-Bit Microcontroller Division uses this standard for all user documentation in order to produce readable, print-friendly coding that enables compatibility among assemblers.

These guidelines are adapted from the code listing style used by the *HCS08 Family Reference Manual.* The data sheet and equate file for the MC9S08GB60 also exemplify the style of the coding standard.

The code standard specifies guidelines concerning:

- Line length limitations
- Acceptable characters and character formatting
- Source program column alignment
- Label formatting
- File and subroutine headers
- Comments formatting

freescale™
semiconductor

This application note demonstrates how the coding standard resolves two issues. Writing code in compliance with this standard helps to ensure readability in printed documentation and prevents incompatibilities that result from syntax differences of assemblers created by different vendors.

Line Length Limitations

This coding standard is unique because the specified line lengths are designed to cooperate with Freescale 8-bit microcontroller documentation templates. Some users may be accustomed to using longer lines and manipulating font size or page orientation for printing code listings. Using the standard ensures neat, readable display.

Code listings use mono-spaced fonts. Because every character takes the same amount of horizontal space on a line, the code's instruction, operand, and comment columns can be lined up to improve readability.

Since these documents are sometimes reduced to a 7 inch by 9 inch page size, the smallest reasonable font size for code is 9 point when printed on an 8.5 inch by 11 inch page. Code listings in Freescale's 8-bit microcontroller documentation use 9 point Courier because it is a mono-spaced font that is readily available on personal computers and Unix workstations. The maximum line length in this normal code paragraph format is 70 characters.

Figure 1 shows an example of a source program that fits within the normal text column. The top two comment lines show column numbers for reference.

```
;        1         2         3         4         5         6         7
;234567890123456789012345678901234567890123456789012345678901234567890
asc2hex:    bsr   ishex       ;check for valid hex # first
            bne   dunA2asc    ;if not just return
            cmp   #'9'        ;check for A-F ($41-$46)
            bls   notA2F      ;skip if not A-F
```

Figure 1. Code Listing Normal Paragraph Style

Figure 2 shows an assembler output listing of the code that was previously shown in Figure 1. This code listing became wider because the information at the beginning of each line was added by the assembler. Using this wider format, listing lines can have up to 93 characters. As in the previous figure, the top two lines show column numbers for reference.

```
;        1         2         3         4         5         6         7         8         9
;234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234
  551 C1D7 AD EA     asc2hex:    bsr   ishex       ;check for valid hex # first
  552 C1D9 26 0A                 bne   dunA2asc    ;if not just return
  553 C1DB A1 39                 cmp   #'9'        ;check for A-F ($41-$46)
  554 C1DD 23 02                 bls   notA2F      ;skip if not A-F
```

Figure 2. Code Listing Wide Paragraph Style

Normally, equate files are shown as source file listings because the assembler output listing does not provide any additional information that is useful to a reader. For this reason, Freescale uses up to 93 characters per line to accommodate more extensive comments in equate files.

Typically, programs are printed in application notes as source programs with up to 93 characters per line. If readers want to examine the object code produced by the assembler, they can download the source file from a Freescale Web site and reassemble it locally to study the listing.

Example code segments to demonstrate various addressing modes and instructions in the CPU chapter of the *HCS08 Family Reference Manual* are sometimes shown as assembler output listings so the reader can see what object code was generated by the assembler. Source listings can use the narrower normal paragraph style while full assembler output listings can use the full page width paragraph style.

Avoid Tab Characters

Always use multiple space characters rather than tab characters in code files. Most code file editors have a configure tab function that instructs the editor to enter a predetermined number of spaces rather than an ASCII tab character. The automatic indenting features that many programmers like to use work similarly.

Because tab characters do not have the same meaning in all word processing programs, tabbing can be problematic. This can lead to alignment problems and confusion when a code file is imported into another document.

Source Program Column Alignment

Source programs will align the label, mnemonic, operand, and comment fields to the following column positions:

- Labels start in column 1.
- Instruction mnemonics start in column 13.
- Operand fields start in column 19.
- Comments start in column 31 except where the operand field extends beyond column 30. In the case of an unusually long operand field, the comment field will be separated from the last character in the operand field by one or two space characters before the semicolon that starts the comment field.

In cases where a label is more than 11 characters long, it should be placed on a separate line above the object code it references. A label on a separate line may include a comment starting in column 31. Shorter labels may be placed on a separate line above the object code to which they refer if the programmer wants the label to stand out. This would most often be done for the label at the top of a subroutine. See Figure 3.

```
;        1         2         3         4         5         6         7
;234567890123456789012345678901234567890123456789012345678901234567890
label:      mne     operand   ;comment
            brset   very,long,operand ;comment can't start in col 31
veryLongLabel:                ;long label on separate line
            nop               ;instruction with no operands
short:                        ;short label may use a separate line
            mne     operand   ;code to which 'short' refers
```

Figure 3. Source Program Column Alignment

Uppercase and Lowercase Characters

The consistent use of uppercase and lowercase characters in programs improves readability and makes them easier to understand. This section provides a set of guidelines for the use of uppercase and lowercase characters in various contexts.

Labels

Labels can include a mixture of uppercase and lowercase characters, but everywhere a label is used it should exactly match the case of the label where it was defined. Although many assemblers can be set to ignore the case of characters in labels, it is poor programming practice to do so.

Instruction Mnemonics

Mnemonics, assembler directives, and pseudo-ops should use lowercase characters. Although most assemblers accept mnemonics and directives in uppercase, it is harder to type shift characters and uppercase characters tend to distract readers.

Instruction set documentation shows mnemonics with uppercase characters to make them stand out. Mnemonics are also shown in uppercase where they appear in paragraph text in data sheets, reference manuals, and other documents. Experienced programmers recognize that the lowercase versions used in code listings are the same to the assembler while being easier to type and to read.

Register and Bit Names

Freescale equate files use uppercase for register and bit names to match the way these names appear in data sheets and other documentation. Using uppercase also helps make these labels stand out compared to other program labels which normally do not use all uppercase.

Freescale equate files define bit names in two ways as shown in Figure 4. Bit manipulation instructions (BCLR, BSET, BRCLR, and BRSET) use a bit number (0–7) to identify which bit will be used as an operand for the instruction. Logical instructions such as AND and ORA use a bit position mask such as %10000000 where the % symbol indicates the value is binary. Since the bit manipulation instructions are much more common than the logical instructions (at least in HCS08 devices), each bit name is equated

to a value from 0 through 7 to identify the location of the bit in a register. The bit name with a prefix of lowercase m is equated to a bit position mask identifying the location of the bit within a register.

```
PTAD:       equ   $00         ;I/O port A data register
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
PTAD7:      equ   7           ;bit #7
PTAD6:      equ   6           ;bit #6
PTAD5:      equ   5           ;bit #5
PTAD4:      equ   4           ;bit #4
PTAD3:      equ   3           ;bit #3
PTAD2:      equ   2           ;bit #2
PTAD1:      equ   1           ;bit #1
PTAD0:      equ   0           ;bit #0
; bit position masks
mPTAD7:     equ   %10000000   ;port A bit 7
mPTAD6:     equ   %01000000   ;port A bit 6
mPTAD5:     equ   %00100000   ;port A bit 5
mPTAD4:     equ   %00010000   ;port A bit 4
mPTAD3:     equ   %00001000   ;port A bit 3
mPTAD2:     equ   %00000100   ;port A bit 2
mPTAD1:     equ   %00000010   ;port A bit 1
mPTAD0:     equ   %00000001   ;port A bit 0
```

Figure 4. Bit Names Defined Two Ways

Labels

Labels use a mixture of uppercase and lowercase characters and underscore characters should be avoided. Some programmers suggest that underscore characters improve readability, while others insist that underscores lengthen labels needlessly or that they can be mistaken for spaces by novice users. Both of these viewpoints have ardent supporters and it is unlikely that this debate can be settled here.

In this coding standard, we stop short of banning underscores altogether, but strongly recommend the selective use of uppercase characters instead of underscores to indicate breaks in multi-word labels. For example, use waitRDRF instead of wait_RDRF or VeryLongLabel instead of very_long_label.

Where a label is defined (that is, where the first character of the label starts in column 1), add a colon (:) after the label. This colon is not part of the label and is not used when the label is used in an operand or expression. Although most new assemblers do not require this colon, some older assemblers used it as a delimiter between the label and the mnemonic or directive field. In those assemblers, the label was not required to start in the first column because the colon identified the preceding characters as a label. In the following example, note the colon after waitRDRF: in the label field but not where waitRDRF is used as an operand.

```
waitRDRF:   brclr RDRF,SCI1S1,waitRDRF  ;loop till RDRF set
```

File and Subroutine Headers

Files and subroutines use title blocks to describe their purpose and to document other important information. These title blocks are sometimes called file headers and subroutine headers. Every file and every subroutine should include a header.

File Header

Figure 5 shows a typical file header (in this case, the file header for the MC9S08GB60 equate file). The basic format and major items of information should be used for all file headers. Some files require slightly different content in the header, depending on the purpose of the file.

```
;*******************************************************************************
;* Title:  9S08GB60_v1r2.equ                    (c) FREESCALE Inc. 2003 All rights reserved.
;*******************************************************************************
;* Author: Jim Sibigtroth - Freescale TSPG
;*
;* Description: Register and bit name definitions for 9S08GB60
;*
;* Documentation: 9S08GB60 family Data Sheet for register and bit explanations
;* HCS08 Family Reference Manual (HCS08RM1/D) appendix B for explanation of equate files
;*
;* Include Files: none
;*
;* Assembler:  Metrowerks Code Warrior 3.0 (pre-release)
;*             or P&E Microcomputer Systems - CASMS08 (beta v4.02)
;*
;* Revision History:
;* Rev #     Date      Who     Comments
;* -----  -----------  ------  ----------------------------------------
;*  1.2    24-Apr-03   J-Sib   correct minor typos in comments
;*  1.1    21-Apr-03   J-Sib   comments and modify for CW 3.0 project
;*  1.0    15-Apr-03   J-Sib   Release version for 9S09GB60
;*******************************************************************************
```

Figure 5. File Header Example

Some items in the file header are required while other items may be omitted for certain kinds of files. In particular:

- Filename: is required and should match the name of the file exactly including any extension.
- The copyright notice is required at Freescale, but other companies may choose to omit it.
- Author: is required because files are often reused years after they were originally written and the author may need to be contacted for help.
- Description: is required, but this should be brief (typically, a single paragraph). More extensive information should be included in separate documentation.
- Documentation: should identify separate detailed documents that explain the file or program in greater detail.

- Include Files: is needed only if this file depends upon the presence of other files which are linked via include directives. The most common cases are a program that needs an MCU equate file or a multi-file project where a main file has include directives for the other files involved in the project.
- Assembler: identifies the brand and version of assembler used to assemble the original file. This is most important when there is a chance the file might be copied into another project where a different assembler is used. Ideally, programs should be written so they can be assembled on any HCS08 assembler.
- Revision History: provides information about when, by whom, and why the file was changed. The original file is revision 0.0 and 1.0 is used for the released version. Whole number changes are used for major changes while digits to the right of the decimal point indicate minor corrections and changes.

Subroutine Header

A subroutine header is similar to a file header but it includes somewhat different information. A representative template is shown in Figure 6. Some items in this template can be removed for the simplest subroutines while other elements require more explanation for the most complex subroutines.

```
;*****************************************************************
;* RoutineName - expanded name or phrase describing purpose
;*  Brief description, typically a few lines explaining the
;*  purpose of the program.
;*
;* I/O: Explain what is expected and what is produced
;*
;* Calling Convention: How is the routine called?
;*
;* Stack Usage: (when needed) When a routine has several variables
;* on the stack, this section describes the structure of the
;* information.
;*
;* Information about routines that are called and any registers
;* that get destroyed. In general, if some registers are pushed at
;* the beginning and pulled at the end, it is not necessary to
;* describe this in the routine header.
;*****************************************************************
```

Figure 6. Subroutine Header Template

Figure 7 shows a completed subroutine header for a complex subroutine that passes information on the stack and uses additional stack space for local variables. Although a much more detailed description is possible, the description in the header is intentionally limited to three or four lines. If more detail is required, it should be provided in separate documentation rather than in the code file.

```
;*****************************************************************
;* GetSRec - retrieve one S-record via SCI1
;*  Terminal program should delay after <cr> to allow programming
;*  recommended delay after <cr> is TBDms, no delay after chars.
;*  Only header (S0), data (S1), and end (S9) records accepted
;*
;* Calling Convention:
;*             ais    #-bufferLength ;# of data bytes + 4 (typ.36)
;*             jsr    GetSRec       ;read S-record onto stack
;*             bne    error         ;Z=0 means record bad
;*;
;*; 'bufferLength' is defined in calling program not in this
;*; subroutine, calling routine must also deallocate buffer space
;*; after processing the information that was returned on the stack
;*;
;*             ais    #bufferLength  ;deallocate buffer space
;*
;*  Returns: all but CCR Z-bit returned on stack (see stack map)
;*    CCR Z-bit = 1 if valid S-record; CCR Z-bit = 0 if not valid
;*    S-record type @ sp+1 (1 ASCII byte) ($30, $31, or $39)
;*    S-record size @ sp+2 (1 hex byte) (# of data bytes 0-31)
;*    S-record addr @ sp+3 (2 hex bytes) (addr of 1st data value)
;*    S-record data @ sp+5..sp+36 (up to 32 hex data bytes)
;*
;*  Stack map... S-record, return, and locals on stack
;*         |                |  <-sp (after local space allocated)
;*  H:X->  | SRecCount      |
;*         | SRecChkSum     |  <-sp (after jsr)
;*         | ReturnAddr msb |
;*         | ReturnAddr lsb |  <-sp (after rts)
;*         | SRecTyp        |
;*         | SRecSize       |
;*  H:X->  | SRecAddr msb   |
;*         | SRecAddr lsb   |
;*         | SRecData 00    |
;*         | SRecData 01    |  etc... (up to 32 bytes)
;*
;*    Data values are read as two ASCII characters, converted to
;*    hex digits, combined into 1 hex value, and stored on stack
;*
;*  Calls: GetChar, PutChar, and GetHexByte
;*  Changes: A, H, and X
;*****************************************************************
```

Figure 7. Complex Subroutine Header Example

The calling convention section is relatively complicated for this subroutine because the user must allocate space on the stack for information that will be filled in and returned by the subroutine. After returning from the subroutine, a BNE instruction is used to check the Z bit in the condition code register (CCR) to determine whether the subroutine successfully retrieved an S-record. After detecting an error or processing the returned data, the calling program must deallocate the stack space that was allocated before calling the subroutine.

The stack usage section is also unusually complex. Since this subroutine manipulates information on the stack, it is important to provide a map of the information to help a reader understand the subroutine. The map shows the large block that was allocated before calling the subroutine (and where specific pieces of information are located in that block), the return address that was stored on the stack as a result of the JSR, and the two bytes of local storage that are allocated from within the subroutine and deallocated again before returning to the calling program.

At the bottom of the subroutine header, the Calls: item lists subroutines that are called from within this subroutine. The Changes: item lists registers that are altered as a result of executing the subroutine.

Figure 8 shows the header for a simple subroutine. This header does not include information about stack usage because it only uses the stack for the return address. The routine does not call any other subroutines, so that information is also not needed.

```
;*****************************************************************
;* GetChar - wait indefinitely for a character to be received
;*  through SCI1 (until RDRF becomes set) then read char into A
;*  and return. Reading character clears RDRF. No error checking.
;*
;* Calling convention:
;*             jsr    GetChar
;*
;* Returns: received character in A
;*****************************************************************
```

Figure 8. Simple Subroutine Header Example

Comments

Comments are a very important part of any program even though they are not translated into object code by the assembler. Comments are used for file and subroutine headers, explanations at the right end of most instruction lines, and for formatting listings so they are easier to read and understand.

The most common form of a comment is a statement that starts with a semicolon in column 31 and extends to the end of the line. For some instructions such as BRSET, the operand field is relatively long and extends beyond column 30. In these cases, leave one or two spaces before the semicolon that marks the start of the comment portion of the line. Avoid comments that simply restate what the instruction does. For example, don't use a comment like this one.

```
            lda   PTAD        ;read port A data
```

Instead, comments should convey information about why the instruction is there or how the instruction relates to the function of the program or the system that contains the embedded microcontroller. This is a better comment.

```
            lda   PTAD        ;check for low on bit 7 (step switch)
```

There is no such thing as self-documenting code. Although it is not necessary to have a comment on every line of code, it should be unusual to see more than a few lines of code in a row with no comments. A whole subroutine with no comments is unacceptable.

Sometimes a comment cannot be abbreviated enough to fit on a line with an instruction. In this case, a longer comment should be placed on a separate line with a semicolon in column 1. Used sparingly, this can improve readability of a program. However, if this is done too often, program instructions can be difficult to see.

```
; In-line comment. Occasionally, a single line comment can be used.
```

In rare cases, it may be necessary to insert a multi-line block comment to explain a particularly difficult passage of code. To make such blocks stand apart from the surrounding code, use a line with only a semicolon in column 1 above and below the block of comment lines as shown here.

```
;
; In-line comment block. On rare occasions, an extended comment
; is needed to explain some important aspect of a program. Each
; line of the extended comment starts with a semicolon. A line
; with nothing but a semicolon in column 1 is used above and
; below the block comment to set it apart from code.
;
```

Comments as Format Elements

Comments or blank lines can help organize a program into logical blocks to improve readability. The subroutine header blocks serve as separators to mark the start of each subroutine. Similar separators are also useful to create visual breaks in other places in programs. For example, a comment block could mark the start of the vector definitions like this.

```
;*************************************************************
;* Start of Vector Definitions
;*************************************************************
            org   Vrti        ;start of vectors

vecRti:     dc.w  rtiISR      ;interrupt service routine for RTI
vecIic:     dc.w  defaultISR  ;handle unused interrupts
 "           "     "           "
```

Leaving a blank line after the org directive helps make the org statement stand out.

Figure 9 shows a block separator for a block of registers in an equate file. It marks the start of the definitions related to a particular MCU module (the internal clock generator in this case).

```
;****  Internal Clock Generator Module (ICG)  ********************************************
;*
ICGC1:      equ   $48         ;ICG control register 1
; bit numbers for use in BCLR, BSET, BRCLR, and BRSET
RANGE:      equ   6           ;(bit #6) frequency range select
REFS:       equ   5           ;(bit #5) reference select
CLKS1:      equ   4           ;(bit #4) clock select bit 1
CLKS0:      equ   3           ;(bit #3) clock select bit 0
OSCSTEN:    equ   2           ;(bit #2) oscillator runs in stop
; bit position masks
mRANGE:     equ   %01000000   ;frequency range select
mREFS:      equ   %00100000   ;reference select
mCLKS1:     equ   %00010000   ;clock mode select (bit-1)
mCLKS0:     equ   %00001000   ;clock mode select (bit 0)
mOSCSTEN:   equ   %00000100   ;enable oscillator in stop mode
```

Figure 9. Block Separator for a Block of Registers in an Equate File

Conclusion

A coding standard sets forth rules for writing and formatting code so that routines written by one person can be easily read or reused by another. Although a typical assembler program allows more freedom, careless use of that freedom can lead to programs that are difficult to understand and may require significant changes to work on a different assembler.

This application note describes a coding standard Freescale uses for assembly language code written for HCS08 user documentation. Following this standard will allow users to easily reuse routines from Freescale documents, thus saving development time for their own applications.

A coding standard is only one part of a successful software development process. Good programming practice also requires other forms of documentation to describe the purpose and operation of application software and systems. The comments within a software program are rarely sufficient to completely explain an application program. Other common forms of software documentation include (but are not limited to) flowcharts and text explanations.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN2111
Rev. 2, 11/2004

freescale™
semiconductor