

Assembly Language Programming Standard For 680X0 Processors

1.0 INTRODUCTION

This document defines a coding standard which you should follow when writing assembly language programs. You will get several benefits by using this standard:

- a) Your code will be well documented;
- b) It will be easier to find bugs;
- c) Code reused on subsequent assignments will be more uniform.

Consistency in coding and documentation are particularly important when you are part of a project team. If you follow the same coding practices as the other members of the team, they will find it much easier to read, understand, and use your code. The software produced by the members of the team will be easier to integrate together. Producing consistent code means following the same coding style, adopting the same naming conventions, and following uniform documentation practices.

Another important reason for adopting a consistent coding style is software maintenance. If a program is to be maintained over a period of time, it is important that you follow a consistent style when making changes and enhancements. Otherwise the program will soon become difficult to read, understand, and modify.

2.0 SUBROUTINE HEADERS

Each program and subroutine must have a header which describes what it does, identifies its limitations, and lists its input and output parameters. The header should contain enough information to allow a programmer to use the subroutine without having to read the code itself.

An outline of the header format and the information it should contain is given on the next page.

Headers For Assembly Language Programming

One program header is required for each assignment. The header will contain:

```
/******  
FILENAME:  
  
STUDENT NAME (AUTHOR):  
  
STUDENT NUMBER:  
  
STUDENT E-MAIL ADDRESS:  
  
LAB SECTION:  
  
ASSIGNMENT NUMBER AND NAME:  
  
PROFESSOR'S NAME:  
  
PURPOSE: be brief but complete  
*****/
```

A subroutine header must be included for **each** subroutine. The header will contain:

```
/******  
HISTORY  
    Date of creation of the routine and the name of the author.  
    For code that undergoes several officially released  
    revisions, this section also contains the date of the  
    revision, the author of the revision, and a short  
    description of each revision.  
FUNCTION  
    Name of the routine and a concise 1 or 2 line description of  
    what it does. Any constraints (limitations of the algorithm  
    used or restrictions on the use of the routine) must be  
    noted.  
DESCRIPTION  
    A pseudocode version of the routine consisting of a HIGH  
    LEVEL, structured, English language description. "High  
    level" means that a line of pseudocode might represent on  
    average 3 to 5 lines of code. "High level" means that the  
    processor's registers are NOT normally mentioned in the  
    code. "High level" means that you EXPLAIN what the code is  
    doing rather than write code-like statements. For example,  
    in pseudocode you do NOT write:  
        FOR Column = 1 to 80:  
            Do something.  
        ENDFOR  
    You DO write:  
        FOR each column on the screen:  
            Do something.  
        ENDFOR
```

Use structured constructs (IF...ELSE... ENDIF, FOR...ENDFOR, WHILE...ENDWHILE, etc.) to describe loops and conditional statements. The keywords used in these constructs (WHILE, ELSEIF, ENDFOR, etc.) must be entirely in upper case.

For every construct, there must be a keyword at the beginning (eg: IF) and another at the end (eg: ENDIF). Statements within the structured construct should be indented by 3 spaces. Using less than 3 does not delimit the construct clearly enough. Using more than 3 moves nested constructs too fast to the right of the page.

INPUTS

A list of the inputs required by the routine. This includes parameters, passed in registers or on the stack, AND variables stored in memory. For each input, you must specify the register or variable name AND provide a short description of what the input represents. It is NOT sufficient just to list the registers containing inputs.

OUTPUTS

A list of all the outputs produced by the routine. This includes registers, variables stored in the memory, and flags or condition codes. For each output, identify the source (the register, variable or flag) AND include a short description of what it represents.

EXTERNAL INPUTS

A list of all inputs received from the keyboard or a file. Describe each external input.

EXTERNAL OUTPUTS

A list of all outputs written to the screen or a file. Describe each external output.

REGISTERS DESTROYED

A list of the general purpose registers destroyed by the routine AND by any subroutines it calls.

*****/

3.0 NAMING CONVENTIONS

One of the keys to writing code that is easily read and understood is to choose meaningful names for variables, subroutines, and symbols or constants. The following rules should help you do this. Note that you should follow the same rules whether you write in assembler or a high level language such as C++ or PASCAL.

1. Choose variable and symbol names that clearly identify what the variable or symbol represents. Similarly, choose subroutine names that describe what the subroutine does.
2. Use NOUNS for variable and symbol names. Use VERBS for subroutine names.

3. In general, do not abbreviate names unless the abbreviation is familiar to most programmers. For example, use "MenuTitle" rather than "MnTtl". However you could use "ctr" for "counter", "MenuPtr" for "MenuPointer", etc. Choose meaningful names that are not excessively long since they could become a typing burden, and can make the code difficult to read.
4. Begin names with a letter. The only exception is a name which for coding reasons must begin with an underscore.
5. Use lower case with the possible exception of the first letter of each word for the name of variables, labels, and subroutines. Where a name consists of several words, highlight the beginning of each word. This will make the name easier to read. You can do this in one of two ways:
 - a) Capitalize the first letter of each word (eg: MenuPtr);
 - b) Separate each word with an underscore (eg: menu_ptr.).

The first method is better because it minimizes the length of the name.

6. Use upper case for ALL the letters in the name of macros and constants or symbols (typically defined using the .MACRO, =, and .SET directives in assembler and the #define statement in C++). Separate individual words in the name with an underscore. For example:

```
FALSE           =      0
ESCAPE          =     27  |ASCII escape characters
MAX_COLUMN      =     80  |Max no of columns on screen
DATA_RATE       =    9600 |Data transmission rate
```

7. Choose MEANINGFUL symbol names. IT is USELESS to write:


```
EIGHT          =      8
```

The name EIGHT defeats the purpose of representing a number by a name to indicate what the number is used for. Here is an example of a symbol name that describes what the number 8 means in some context:

```
BITS_PER_BYTE  =      8
```

4.0 CODING STYLE

1. Make your subroutines short. On average, you should have no more than 50 executable statements per subroutine and definitely no more than 100.

2. Never put more than 80 characters on a line. Every line should be fully visible on a standard video display terminal. A line in a listing should have no more than 132 characters so that it can be printed on one line on a 132-column printer.
3. Define symbols at the beginning of a file. Better yet, put the symbol definitions, especially those used in more than one file, in a separate INCLUDE file.
4. Declare variables in a separate data area at the beginning of a file after symbol definitions but before any code. This puts the data for the subroutines in one spot and minimizes forward references.
5. Don't use numbers (with the possible exception of 0 and 1) in the body of your code. Define a symbol for each number and use the symbol in the code.
6. Make the assembler do any arithmetic at assembly time rather than doing it yourself especially when it makes the code easier to understand. For example:

```
MAX_CHAR_ON_SCREEN = 80*25 |No of char on the screen
```

The definition above is much better than:

```
MAX_CHAR_ON_SCREEN = 2000 |No of char on the screen
```

There are two reasons for this. First, the assembler is much less likely to make arithmetic errors than you are. Second, the first statement immediately tells the reader exactly what size of display the program is written for. Its 80 columns, not 132. Its 25 rows, not 24, 43 or 50.

7. Use lower case for processor instructions and registers. Use upper case for assembler directives and operators. For example:

```
Value:    .LONG    0x12345678    |Variable declaration
DISPLACE =    0x24          |Symbol declaration
:
:
        move.w    d1, (DISPLACE, a1, d2.w)
        add.l     d4, (Value)
```

8. Put a comment above each "block" of code in a subroutine that carries out a specific task. The comment should explain what the block of code does. Use proper English, and format the comment as follows:

```
|
| This comment explains the block of code that follows.
|
```

9. An in-line comment is used to explain what some instruction is doing or to explain the purpose of a variable or symbol that is being declared. As the name implies, an in-line comment appears on the same line as the instruction or declaration. Because of the limited space, you can make in-line comments terse.
10. Use a comment to explain the purpose of EVERY variable you use in your program when you declare it. The comment should either precede the declaration or be in-line. Here are examples:

```

|
| Prompt asking user to type in the second number
|
Prompt:      .ASCIZ          "Enter the next number:"

```

11. Avoid over-commenting and redundant or USELESS comments. In the examples below, the in-line comments are USELESS because they provide NO MORE information than the instruction itself:

```

      movem.l      d1,-(a7)  |Put d1 register on stack
      adda.w      d4,a1     |Add d4 to a1

```

12. For instructions that can operate on bytes, words, or long words, always specify the size extension (.b, .w, or .l). You always indicate explicitly what you want to do, and the reader never has any doubt about the intended size of the operands.
13. The branch instructions (Bcc and BSR) for 16-bit members of the 680x0 family (68007, 68000, 68010, etc.) support 8-bit and 16-bit displacements. To explicitly specify the size of the displacement, Motorola used .s for short (8-bit) and .l for "long" (16-bit). The 32-bit members of the 680x0 family support branches with 8-, 16-, and 32-bit displacements. To accommodate 32-bit replacements, Motorola revised the assembler syntax to .b (8-bit), .w (16-bit), and .l (32-bit). There is usually an assembler directive (NOOLD for the Avocet assembler) to distinguish between the two options.

If the assembler supports it, use the new style in preference to the old when you want to specify the size of the displacement explicitly. Note that you don't have to specify the size. For backward references, the assembler will calculate the size and reserve enough bytes for it. For forward references, the assembler will typically reserve a word for the displacement.

14. If the assembler supports it, use the new or 32-bit style of specifying memory addresses rather than the old or 16-bit style. Examples of both styles are given below:

```

move.l    0x12(a1,d4.w), 0x1234(a3)    |Old style
move.l    (0x12,a1,d4.w), (0x1234,a3)|New style

```

In the new style, all components of the addressing mode are within parentheses. The components are separated from each other with commas.

5.0 EXAMPLE

```

/*****
HISTORY
  11 Jun 91  Written by Leopold Beaudet.
  20 May 02  Modified (comments & directives) for gnu gas
             compatibility by Shawn McBride
FUNCTION
  PrintMsg
  Sends a null-terminated string whose address is in a0 to the
  output port.
DESCRIPTION
  WHILE the next character of the string NE 0:
    Transmit the next character of the string.
  ENDWHILE
INPUTS
  a0.l  Pointer to Null-terminated string.
OUTPUTS
  None.
EXTERNAL INPUTS
  none.
EXTERNAL OUTPUTS
  String which a0.l points to will be displayed on screen.
REGISTERS DESTROYED
  None.
*****/

PrintMsg: movem.l  a0/d0,-(a7)
Pm01:     move.b   (a0)+,d0
          beq.s    Pm02      |Null terminator found
          bsr     ConsoleOut
          bra.s    Pm01      |More bytes left in message
Pm02:     movem.l  (a7)+,a0/d0
          rts

```