Assembly Language Coding Standards

Ken Clowes (kclowes@ee.ryerson.ca)

November 11, 2000

Contents

1 Introduction

This document describes coding standards for assembly language programs. While the standards described could be used with any assembly language, the examples assume the DECUS variant[1] of the 6811 assembly language (i.e. the format used in the third year microprocessor course—ELE 538—at Ryerson.)[2] Even more specifically, we describe writing a library module of functions.

Some general background to coding standards in any language is described in "General Coding Standards" [Clo].

2 Public Documentation

We use the term public documentation to describe the information a user of a module needs to know about what can be done with the module and how to use its features. Implementation details—how implementation works—should not be public.

Each source code file is a module containing one or more subroutines. The entire module should have a public comment describing its overall use and each subroutine should be described. We use the convention that any comment on a line beginning with `;;` is a public comment.

For example, suppose a module called `strings.asm` contains several functions similar to the C functions in the standard `strings.h` library. The public documentation could look something like this using traditional comments.

```
;;
;; The string MODULE contains a collection of string routines
;;
;; The SUBROUTINE strlen determines the length of a null-terminated string...
;;
;; ENTRY CONDITIONS:
;;    Register X -- the starting address of the string
;;
;; EXIT CONDITIONS:
;;   Accumulator B contains the length of the string
```

[1]Written by Alan R. Baldwin at Kent State University and placed in the public domain.
[2]Appendix B describes other versions of 6811 assembly language.

This informal kind of public comments may be all that is required. However, I suggest that each subroutine be more formally described with each of the following features:

Name: Obviously every subroutine has a name that should evoke its meaning.

Description: A brief description of what the subroutine does.

Parameters: What are the parameters to the subroutine and how are they passed.

Return value(s): What (if anything) does the subroutine return and how does the caller obtain these values.

Side effects: What other side-effects (such as the modification of registers or global variables) are done.

I have written a simple script—asmdoc[3]—which extracts these structured comments and creates a cross-referenced HTML file suitable for Web viewing.

Keywords beginning with the @ character are used to structure the comments. For example, some of the public comments for the strings module are:

```
;; @module strings
;; This module contains a collection of string routines.
;;
;; @name strlen
;; Determines the length of a null-terminated string.  If
;; the string is less than 256 characters in length, the correct
;; length is returned in Accumulator B and the Carry bit in the
;; Condition Code register is cleared.  Otherwise, 255 is returned
;; and the C bit is set.
;;
;;
;; @param Register X -- the starting address of the string
;; @return AccB -- the length of the string (if less than 256
;;         characters).
;;         <dd>CC -- Carry bit Set if length > 255; else Cleared
```

[3]The asmdoc tool is described in Appendix A.

```
;; @side none
;;
;; @name strcpy
;; Copies a null-terminated string.
;;
;; @param X source string
;; @param Y destination string
;; @return nothing
;; @side Registers A, X, Y are modified.
```

The `asmdoc` tool converts these to HTML which can be viewed in a browser; Figure 1 is a screen shot of some of the generated documentation.

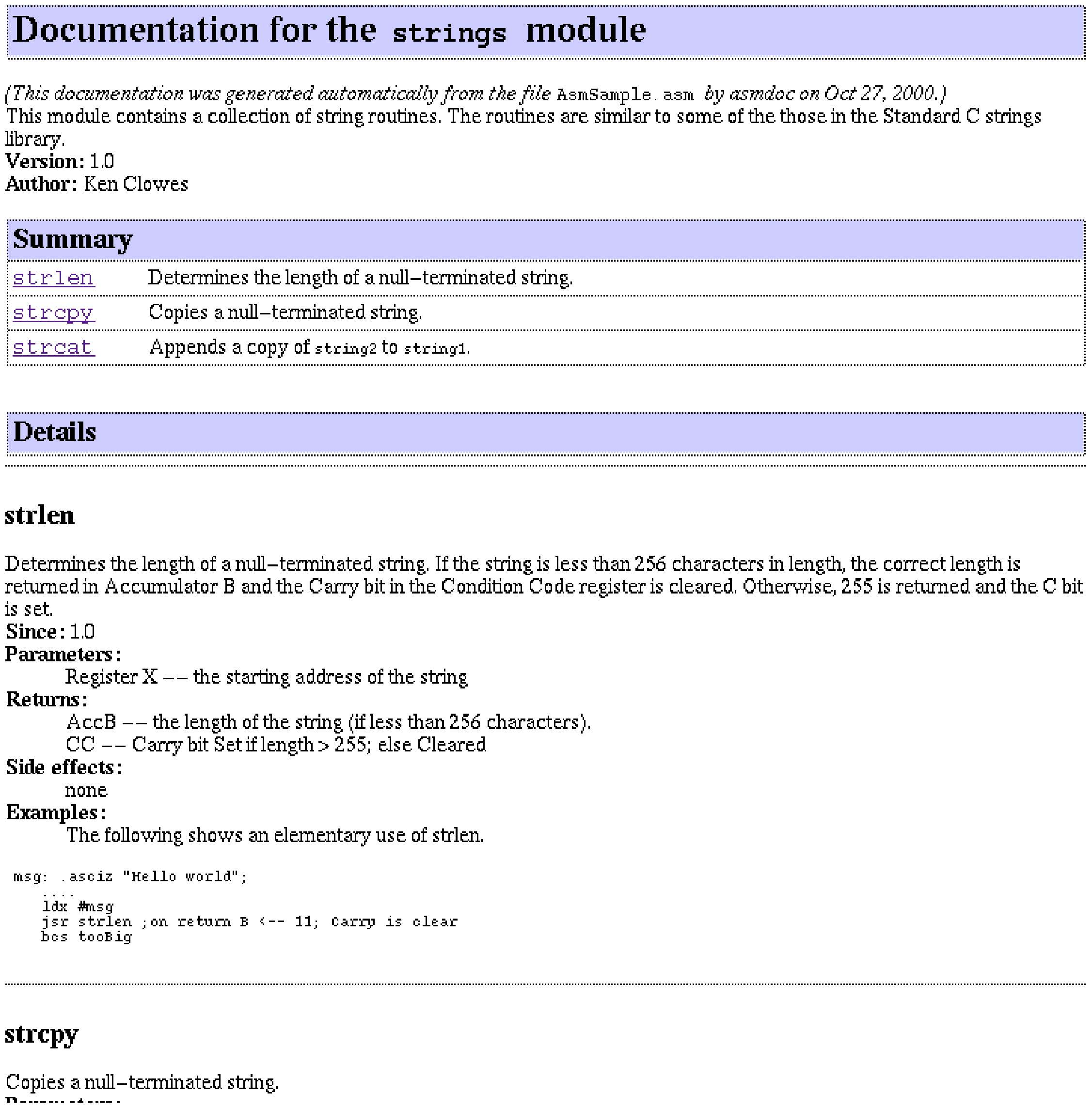
It is useful to write the public documentation (and generate a more readable formatted version) before writing code. Documenting the interface precisely forces you to think about what you really want the code to do and you may detect some ambiguities or inconsistencies in the public specifications.

For example, the interface to `strlen` is inconsistent with the one for `strcpy`. In particular, the `strlen` is documented as having no side effects (apart of course from the return values in Accumulator A and the Condition Code register); however, `strcpy` is documented as modifying the registers A, X and Y[4]. Once this inconsistency is detected, the programmer should decide which convention to use and modify the documentation so that all interfaces are reasonably consistent.

3 Assembly language coding

The best programmers are lazy—they want to be able to read, maintain, modify and ensure the correctness of their software with as little effort as possible. Curiously, writing source code that allows the programmer to be lazy in the future (reading, testing, maintaining) requires hard work.

[4]The implementor of `strlen` is probably thinking that the registers used by the subroutine can be pushed onto the stack before use and their values can be restored by popping them off before returning, hence avoiding side effects. The implementor of `strcpy`, on the other hand, is probably thinking this is the caller's responsibility and the subroutine will work faster without the pushing and popping. There are valid arguments for either convention, but there is little justification for not using a single convention for closely related routines in the same module.



Documentation for the `strings` module

(This documentation was generated automatically from the file `AsmSample.asm` *by asmdoc on Oct 27, 2000.)*
This module contains a collection of string routines. The routines are similar to some of the those in the Standard C strings library.
Version: 1.0
Author: Ken Clowes

Summary

strlen	Determines the length of a null–terminated string.
strcpy	Copies a null–terminated string.
strcat	Appends a copy of `string2` to `string1`.

Details

strlen

Determines the length of a null–terminated string. If the string is less than 256 characters in length, the correct length is returned in Accumulator B and the Carry bit in the Condition Code register is cleared. Otherwise, 255 is returned and the C bit is set.
Since: 1.0
Parameters:
Register X –– the starting address of the string
Returns:
AccB –– the length of the string (if less than 256 characters).
CC –– Carry bit Set if length > 255; else Cleared
Side effects:
none
Examples:
The following shows an elementary use of strlen.

```
 msg: .asciz "Hello world";
    ....
    ldx #msg
    jsr strlen ;on return B <-- 11; Carry is clear
    bcs tooBig
```

strcpy

Copies a null–terminated string.

Figure 1: Screen shot of formatted documentation

We now examine the nitty-gritty details of writing assembly language code that is readable, maintainable and safer (than it otherwise would be) and that allows the programmer to be lazy in the future.

3.1 Numbers

Assembly language programmers are only a small step away from the bottom level binary machine language that the computer actually interprets. The machine only "sees" bit patterns, but the assembly language programmer can specify a particular pattern in various ways.

Let's consider a simple example. Suppose the underlying bit pattern the programmer wishes to express is "`01000011`". This sequence of bit values can have any number of higher level meanings. Some possibilities are:

- It represents the decimal number `67`.
- It represents the opcode for a machine language instruction. For example, we usually think of opcodes in hexadecimal (which is how they are described in data sheets) and the 6811 instruction `COMA` has the opcode `0x43`.
- It represents the ascii code for the letter 'C'.
- It represents something that the programmer really thinks of as a bit pattern; in this case, it can be expressed in hexadecimal(`0x43`), octal (`0o103`) or binary (`0b01000011`) notation.

All of the following assembly language instructions generate the identical machine language code (`0x8643`). However, they express the bit pattern "`01000011`" in different ways that more closely resemble the programmer's intent and are augmented with additional private comments that make the intent even more explicit.

```
ldaa #67        ;expected average percent grade (rounded up)
ldaa #'C        ;expected average letter grade
ldaa #0x43      ;Opcode for the COMA 6811 instruction
ldaa #0b01000011 ;DEVfoo cntrl: intrpt, pulse handshake, POS logic
```

3.2 Symbolic Constants

The previous examples stress that numbers should be expressed in a format closest to the programmer's abstract idea of the number. This alone, however, is often not sufficient. Rather, important constants should be given symbolic names that reflect their meaning.

As a simple example, the code above should be re-written by first defining the values of symbolic constants (with meaningful names) as follows:

```
EXPECTED_AVG_PERCENT_GRADE = 67; (rounded up)
EXPECTED_AVG_LETTER_GRADE = 'C
OPCODE_COMA = 0x43
INT_ENABLE = 0b01000000
PULSE_HAND = 0b00000010
POS_LOGIC  = 0b00000001
CTRL_IE_POS_PULSE = INT_ENABLE | PULSE_HAND | POS_LOGIC
```

The instructions can then be written so that their meaning is self-evident:

```
ldaa #EXPECTED_AVG_PERCENT_GRADE
ldaa #EXPECTED_AVG_LETTER_GRADE
ldaa #OPCODE_COMA
ldaa #CTRL_IE_POS_PULSE
```

Once again, all of these instructions produce the identical machine language (`0x8643`).

Let's look at this issue from the opposite point of view. Suppose that the machine language for a section of code is `0xCC02EEBD7123`. The machine language instructions can be disassembled to produce:

```
ldd #0x02EE
jsr 0x7123
```

It is conceivable (though unlikely) that this is what the assembly language programmer wrote.

Suppose we also know that the subroutine at address `0x7123` simply delays for the number of bus cycles specified in Accumulator D. We also assume that the bus speed is 1 MHz (hence a bus cycle lasts 1 μsec). It is now plausible that the programmer wrote:

```
        ; delay for 750 microseconds
    ldd #750
    jsr delay
```

The program is now more readable. It is also safer since the assembler will figure out the correct address of the `delay` subroutine. Had the programmer really written `jsr 0x7123`, she would have to check that the address was correct when any changes were made to the source code; furthermore, if she mis-typed "7123" as "7132", the assembler would not detect the error. With the symbolic name, no complex address calculations are required and if she had mis-typed "delay" as "dealy", the assembler would signal an "undefined symbol" error message.

The remaining problem with the source code is the appearance of the "magic number" 750 embedded into an instruction. To see the potential for mischief, suppose that 750 is embedded into several `ldd #750` instructions and that sometimes 750 means the delay time in microseconds but on other occasions it represents the hourly wages in cents (i.e. $7.50/hour) for a hamburger flipper. If we want to change all the 750 μsec delays to 800 μsec, we have to be very careful. The solution, of course, is to use symbolic constants as follows:

```
    MAC_WAGES = 750 ;units = pennies per hour
    DELAY = 750     ;units = microseconds
        .
        .
    ldd #DELAY
    jsr delay
        .
        .
    ldd #MAC_WAGES
    jsr payMe
        .
        .
    ldd #DELAY
    jsr delay
        .
        .
        .
```

It is now very simple to change all the delay times and not change the restaurant wages by editing a symbol definition. This makes a lazy programmer happy (they don't even have to understand the source code; reading the comments for the symbol definitions is sufficient) and the modifications are safer.

But we can do even better. If the program is to run on a microprocessor with a 2 MHz bus, all the constants that depend on a 1 MHz bus will have to change. Once again we can satisfy the lazy programmer and make modifications more safely with the following:

```
CYCLES_PER_MICRO = 1
DELAY_IN_MICROS = 750 ;microseconds
DELAY_PARAM = CYCLES_PER_MICRO * DELAY_IN_MICROS

ldd #DELAY_PARAM
jsr delay
```

3.3 Names

Assembly language is a symbolic language and a competent programmer chooses sensible names for constants, subroutines, variables and labels.

One important aspect of names (especially ones that are exported) is the possible implementation specific limitations on the length of names and their case sensitivity. For example, some assemblers are case insensitive, but others are not. Hence it is a good rule of thumb to avoid names that differ only by the case of letters. This rule does not mean that you should use only upper-case or only lower-case names. You can and should use different cases as a visual clue to the reader of the source code.

Some assemblers or linkers limit the meaningful length of symbols to 8 characters, so it is a good idea (for portability) to ensure that all symbols are unique in their initial 8 characters[5].

[5]Hence symbols like `EXPECTED_AVG_LETTER_GRADE` and `EXPECTED_AVG_PERCENT_GRADE` may cause problems. In the DECUS environment, the symbols are treated as distinct only within a module; if they were global, their names would conflict because they do not differ in the first 8 characters.

3.3.1 Symbolic constants

The importance of using symbolic constants instead of embedding magic numbers into source code has already been discussed. But what kind of names should be used? As always, the most important rules are to use "common sense" and be consistent.

My preference for symbolic constants is to give them clearly descriptive names using upper-case letters. For multi-word names, I prefer using the underscore character (_) to separate the words.

In the case of names that are defined by manufacturers' data sheets, it is preferable to use the established name rather than inventing your own. For example, use the name `ADCTL` for the 6811's A/D control register rather than something like `AD_CONTROL_REGISTER`.

Note also that in the case of built-in device registers (usually memory mapped in the memory space starting at 0x1000 for the 6811), there are two common ways to address a register: either you can use the absolute address or use indexed addressing with IX containing the base address of the register area. Since indexed addressing is the most common, my preference for address equates is:

```
REGBAS = 0x1000            ;Base address of I/O register block
ADCTL  = 0x30              ;Offset to the A/D control register
ADCTL_ABS = REGBAS + ADCTL ;Absolute address of A/D control reg.

ldx #REGBAS
  ;The following two instructions do the same thing:
staa ADCTL,X     ;indexed addressing ==> 2-bytes, 4 cycles
staa ADCTL_ABS   ;extended addressing ==> 3-bytes, 4 cycles
```

3.3.2 Subroutine and variable names

Subroutines and variables should be given descriptive names; this is especially important for those that are exported (i.e. are described in the public documentation of the interface).

My preference is to use either short lower case names (especially for subroutines that are similar to standard C library functions such as `strlen`). In the case of more obscure subroutines with multi-word names, I often use upper case to highlight the beginning of each word (such as `DrawRectangle`).

Names of subroutines or variables that are meant to be accessed from either assembly code or higher level languages such as C must begin with an underscore (_) character. (For example, the a subroutine named `strlen` could not be called from C but one called `_strlen` could be.)

3.3.3 Labels

Labels should be used for the targets of all branch statements. (i.e. avoid statements with explicit relative addressing such as `bra .+3`.)

The organization of assembly language programs should follow structured techniques rather than unorganized "spaghetti code". For example, consider the following pseudo-code design of a program fragment:

```
#define FOO 5
if (AccA == FOO) {
   IX++;
   AccB++;
}
IY++;
```

When translated into assembler, there has to be a conditional branch around the "then clause" and it seems reasonable to label the target of the branch as "endif" as shown below:

```
   FOO = 5
   cmpa #FOO
   bne endif
      inx
      incb
endif:
   iny
```

Unfortunately, if there is more than one "if statement", the "endif" label cannot be re-used. One way around the problem is to use labels like "`endif_1`", "`endif_2`" and so on. For larger modules, I prepend an abbreviation of the subroutine name obtaining labels like `len_ef1`. It is also permissible, of course, to use more meaningful labels for branch targets such as "`bad`" or "`oops`" or "`done`".

3.4 Formatting

Precise standards for formatting are an individual or organizational choice. Whatever standards you adopt should aid the reader of the source code and be applied consistently. In particular, the mere fact that your source code is legal does not justify ugly, inconsistent formatting. Avoid writing code like:

```
 FOO = 5
      cmpa       #FOO
   bne   endif
                inx
incb
   endif:iny
```

I recommend that you indent code (or use horizontal space) in some sensible fashion (but avoid tabs, use spaces) and limit the length of lines to 80 characters or less.

If you use (x)emacs, you can add the following to the ~/.emacs file:

```
; This adds additional extensions which indicate files normally
; handled by asm-mode
(setq auto-mode-alist
      (append '(("\\.asm$"  . asm-mode)
                )
              auto-mode-alist))
; Use auto-fill-mode (minor mode) in asm-mode
; Can be annoying...you may wish to turn it off
(add-hook 'asm-mode-hook 'turn-on-auto-fill)

;show line-numbers in the mode-line
(setq line-number-mode t)
```

3.5 Private implementation comments

Assembly language has far fewer structured flow control and data typing features than high level languages do. Consequently, it is common for assembly language programs to be commented in far more detail than a high level language would be. Remember, however, that these detailed implementation comments are not meant to be read by someone who merely wants to use a subroutine you have written. All the information they require should be in the public comments.

Although comments may be quite detailed, keep in mind that they are only meant to be read by an assembly language programmer who wants to understand or modify your code. Do not insult their intelligence with useless comments like:

```
inca ; Increment Accumulator A by one.
```

3.6 A simple example

We can put some of the ideas together with a simple example. Suppose Accumulator A contains the binary representation of a decimal digit (i.e. a number between 0–9) and we want to transform it into the character representation of the digit.

A straightforward, but poor, way of doing this would be:

```
adda #0x30  ;even worse would be adda #48
```

Much better, of course (at least if you have been reading carefully) would be:

```
adda #'0
```

However, we should also check that the initial value in Accumulator A is valid. We should also specify what to do if it is not valid; perhaps, we could return the character '?' if the digit were invalid. We could transform the whole sequence into a subroutine, obtaining:

```
ILLEGAL_DIGIT_RETURN = '?
digtoa:
       tsta
       bmi bad  ;the digit must be >= 0
       cmpa #9
       bhi bad  ;it also must be <= 9
         adda #'0
         rts
bad:   ldaa #ILLEGAL_DIGIT_RETURN
       rts
```

Of course, public documentation of the `digtoa` subroutine interface should have been written first, but we leave this as an exercise.

4 Organization

Each project should have its own directory. The directory should always contain a `README` file that briefly describes what the project is about and the important files in the directory.

There should also be a `Makefile`. Normally, the default target for `make` should create any object modules or `.s19` files and documentation files required by the user.

Other common targets are:

clean: Removes generated files.

doc: Creates documentation files.

archive: Creates an archive of source files.

test: Runs tests on the software.

4.1 Assembly language code organization

Modern assemblers and linkers allow the programmer to be less concerned with the absolute address of entities when writing their programs and allow allow logically distinct portions of the assembly code to be organized into distinct areas or segments that the linker can deal with.

For example, I tend to organize even the most trivial program into at least two sections: one for code and another for data. A general template (excluding public comments) looks like:

```
; Symbolic constants
  definitions of symbol constants go here

.area DATA
  variables go here

.area _CODE
  actual instructions go here
```

One advantage of doing this (even in trivial programs) is that you can set the actual absolute address of each of the segments at link time. On simple programs, I often set the start of the data area to 0x6000 and the start of the code area to 0x6200. By placing important variables at the beginning of the data segment, I can view many of them just by examining a memory dump of 0x6000–0x6010.

Another advantage is the ability to intermix `.area DATA` and `.area _CODE` directives. For example, if you have several subroutines where some use global memory references common to all of them and some also use static references that are private, you can write code like:

```
    .area DATA
      common variables go here

  ;foo starts here...
    .area DATA
      foo variables go here

    .area _CODE ;for the foo routine
foo:
      actual instructions for "foo'' go here

  ;bar starts here...
    .area DATA
      bar variables go here

    .area _CODE ;for the bar routine
bar:
      actual instructions for "bar'' go here
```

5 A Real Example

I have written a real example illustrating many of the points discussed here. This example also illustrates the use of makefiles and how to test software.

The example is self-documenting. To obtain a copy and try it out do the following:

1. Create a new directory and change to it. (For example `mkdir coding; cd coding`.)
2. Copy the example archive and unpack it with the commands:

```
        cp ~kclowes/public/CodStdEx.tgz .
        zcat CodStdEx.tgz | tar xvf -
```

3. Use Netscape to examine the directory you are in. Click on the file called `README` and read it.
4. In the shell, invoke the command `make`.
5. Hit the reload button in Netscape to see the new directory listing.
6. There are lots more files now. Browse some of them (especially the `.html` ones; a good starting point is `test1.html`.)
7. Once you get the general idea of what is going on (but before your eyes glaze over), run the command `make ex` in the shell, read the newly created file `exercises.html` and do what you can...

A The asmdoc tool

The `asmdoc` tool is a simple little program (actually a perl script) that translates public comments as described here into formatted and cross-referenced HTML files.

To use `asmdoc`, add public comments to an assembly language file, say `foo.asm` and invoke the command `asmdoc foo.asm`. A HTML file with the same base name (`foo.html` in this case) will be generated and can be viewed with a browser such as Netscape.

All public comments must be on lines that begin with two semi-colons (;;).

Some special tags are used and begin with the `@` character. In particular, the first public comment should be:

```
;;  @module Your_module_name
```

Until the next special tag word (one starting with @), the following public comments are basically treated as paragraphs; a blank public comment line separates paragraphs. For example:

```
;;This is the beginning of a paragraph. Here is the second
;;sentence. <B>Boring</B> stuff...but the next paragraphs are elegant.
;;
;;Fourscore and seven years ago our fathers brought forth on this
;;continent a new nation, conceived in liberty and dedicated to the
;;proposition that all men are created equal.
;;
;;Now we are engaged in a great civil war, testing whether that nation
;;or any nation so conceived and so dedicated can long endure. We are
```

The HTML browser will reformat the paragraphs so that they are justified on the screen so don't worry too much about the visual formatting of the comments in the source code. If you know HTML, you can add your own HTML directives. For example, the `<B>Boring</B>` uses HTML markup commands to display the word "Boring" in bold.

While you can add any paragraphs you want after the `@module` tag (even the Gettysburg Address), you should write zero or more paragraphs that describe the module in general terms.

You can also use other special tags like `@version`, `@author` and `@example`. (see below for their precise meanings) in the module section if you wish.

Following the module section, each public subroutine should be described. The public documentation of a subroutine must begin with the special tag `@name` followed by the name of the subroutine being documented. For example:

```
;;  @name strlen
```

You can then include any number of free form paragraphs that describe the subroutine. The very first sentence should be short and will be used in the summary section that `asmdoc` generates to briefly describe each documented subroutine and provide a link to the detailed documentation. Consequently, there must be at least one sentence following a `@name` tag.

Following the general description of the subroutine you should use the `@param` tag for each (if any) parameters passed to the routine. Next, the way any results are returned should be described with the `@return` tag. If there are any additional side effects (such as the modification of other registers or global variables), they should be commented with the `@side` tag.

You may also want to include examples of use; use the `@example` tag to introduce them. Since examples usually include a few lines of assembly that we do not wish the browser to format these lines as a paragraph. The HTML "pre" directive (for pre-formatted) tells the browser to render the lines between the `<PRE>` and `</PRE>` exactly as you typed them. (Of course, `asmdoc` will remove the leading semi-colons.) For example:

```
;; @example
;; The following shows an elementary use of strlen.
;; <PRE>
;; msg: .asciz "Hello world";
;;    ....
;;    ldx #msg
;;    jsr strlen ;on return B <-- 11; Carry is clear
;;    bcs tooBig
;; </PRE>
```

A.1 Detailed description of asmdoc tags

@module: The module tag is required. There must be only one in the source code file and it must appear on the first public comment line. The syntax for it use is:

```
;;     @module  module_name
  Followed by paragraphs describing the module
  and possibly with the tags: @version, @author, @example
  The module section ends with the first @name tag.
```

@name: Every documented subroutine begins with the `@name` tag:

```
;; @name  subroutine_name
  Description follows...
```

@version: If used, there should only be one `@version` tag and it should be in the module section:

```
;; @version  version_name
```

@param: Each parameter is described with this tag:

```
;; @param  name and description for first parameter
;; @param  name and description for second parameter
         etc....
```

@since:

@author:

@return:

@side:

@example:

B Assembler conventions

There are several different syntaxes for assemblers. Most of the differences involve assembler directives.

Table 1 shows some of the differences between the DECUS and Motorola assemblers commonly used at Ryerson.

Directive	DECUS example	Motorola example
Equates	`FOO = 123`	`FOO EQU 123`
Hex numbers	`0x1A`	`$1A`
Defining bytes	`.db 123`	`FCB 123`
Defining words	`.dw 123`	`FDB 123`
Reserving bytes	`.ds 5`	`RMB 5`

Table 1: Different assembler conventions

It should not be too difficult to write a text transformation program to convert from one style to another. Any volunteers?

B.1 A Structured Assembler

Traditional assemblers differ in conventions, but I believe a structured assembler could make the assembly-language programmer's life easier.

One aspect of most assembly languages I am aware of is the lack of support for structured flow control. This seems to be the "natural" consequence of the lack of this feature at the machine language level; at this level the only way to change the program counter from its default behavior is with the conditional or unconditional "goto" mechanism. But this basic fact does not imply that assembly language programs should be designed using unstructured flow control nor that an assembly language cannot offer some help for such designs. Let me be absolutely clear that the resulting assembly syntax is still assembler: every syntactical convention of a traditional assembler is still available and every line of source code translates into a single machine language instruction or traditional assembler directive or label. It is an assembler, not a very primitive high-level language.

I have not yet written this kind of structured assembler (but I hope a student may consider the implementation for a senior project).

I can briefly explain the concept with some simple examples.

Consider first the simplest structured flow control statement: the if statement. We might design in pseudo-C something like:

```
  #define FOO 5
  if (AccA == FOO) {
     RegX++;
     AccB++;
  }
  RegY++;
```

This can be translated into assembler:

```
   FOO = 5
   cmpa #FOO
   bne endif
      inx
      incb
endif:
   iny
```

The translation is straightforward except for coming up with the label `endif`. (Sure, some assemblers allow local target label names that can be reused. While this is better than nothing, the syntax is often obscure and the fundamental problem of forcing the programmer to come with a label is not addressed.)

With a structured assembler, the assembly language code would be:

```
FOO = 5
cmpa #FOO
if == {
   inx
   incb
}
iny
```

In this case the curly braces delimit the extent of the "then clause". The programmer, however, is relieved of the task of generating a label for the first instruction following the then-clause; the structured assembler will do this for her.

There are two other advantages:

1. The "sense" of the original design is maintained. The structured assembler will translate the `if ==` instruction into the machine instruction corresponding to `bne endif`.

2. The use of braces also allows clear indentation (possibly automatic) to increase the readability of the code.

Let's now consider the next more complex flow control structure—the if...else structure. The pseudo-C design is:

```
#define FOO 5
if (AccA == FOO) {
   RegX++;
} else {
   AccB++;
}
RegY++;
```

The traditional assembler implementation is:

```
   FOO = 5
   cmpa #FOO
   bne else
      inx
      bra endif
else:
      incb
endif:
   iny
```

With a structured assembler, we would write:

```
   FOO = 5
   cmpa #FOO
   if == {
      inx
   } else {
      incb
   }
   iny
```

Once again, the transformation from the structured syntax to traditional forms is straightforward with labels added for the beginning of the "else clause" and the continuation following the "endif". Note that it is still assembler with one machine instruction per assembler instruction; as before, the `if ==` structured instruction is transformed to a `bne else` traditional instruction. Of course, there also has to be an unconditional branch around the "else" part at the end of the "then" clause. In effect, the structured instruction `} else{` is transformed into the necessary branch.

```
clra
while ( cc ) {          ; translated to bcs endWhile
   ldx #"Hello"        ; read-only static string created
   putstr()             ; translated to jsr putstr
   inca
}                       ; translated to bra ...  and endWhile label added
```

One advantage of using `putstr()` instead of `jsr putstr` is the ability to translate the line either into a `jsr` instruction or to expand it in-line (with flags in the assembler or with pragmas).

References

[Clo] Ken Clowes. General Coding Standards. file: `CodingStdGen.ps`.