A.M. 100 DESIGN & DOCUMENTATION STANDARDS

Written by
Jens K. Dill, Robert W. Hopsch,
and Douglas F. Dixon

With extensive help from
John Farrar, Rick Harrington, Steve Lewis,
Craig Mathias, David Meyer, Marty Michel,
David Notkin, Andries van Dam, and others
# TABLE OF CONTENTS

## FOREWORD

---

## INTRODUCTION TO THIS MANUAL

---

## WHY READ THIS HANDBOOK

---

## SUMMARY OF TCP-DGN DESIGN

---

## PHYSICAL DOCUMENTATION COMPONENTS

### 2.1 ENGLISH LANGUAGE DESCRIPTIVE (FLD)

---

### 2.1.1 USER-ORIENTED INFORMATION

---

### 2.1.2 MODULAR SPECIFICATIONS

---

#### 2.1.2.1 NAME

---

#### 2.1.2.2 FUNCTION

---

#### 2.1.2.3 CALLED BY

---

#### 2.1.2.4 ROUTINES CALLED

---

#### 2.1.2.5 PARAMETERS

---

#### 2.1.2.6 CALLING SEQUENCE

---

#### 2.1.2.7 ENTRY POINTS

---

#### 2.1.2.8 NORMAL EXITS

---

#### 2.1.2.9 BACK CONDITIONS

---

#### 2.1.2.10 INPUT AND OUTPUT

---

#### 2.1.2.11 GICAL ASSUMPTIONS AND SIDE EFFECTS

---

#### 2.1.2.12 STORAGE UTILIZATION AND DATA

---

#### 2.1.2.13 ALGORITHM

---

#### 2.1.2.14 DESIGN CONSIDERATIONS

---

#### 2.1.2.15 COMMENTS

---

#### 2.1.3 SUB HISTORY

---

#### 2.1.4 DESIGN DESCRIPTION

---

## INTERNAL DOCUMENTATION

### 2.2.1 BLOCK COMMENTS

---

### 2.2.1.1 REGISTER USAGE

---

### 2.2.2 INLINE COMMENTS -- GENERAL DESCRIPTION

---

### 2.2.3 STATEMENT COMMENTS

---

### 2.2.4 FLOW OF CONTROL

---

#### 2.2.4.1 GICLCS AND LABELS AND STRUCTURED PROGRAMMING

---

#### 2.2.4.2 INDENTATION

---

### 2.2.5 CONDITIONAL STATEMENTS

---

#### 2.2.5.1 THE IF CONSTRUCT

---

#### 2.2.5.2 THE CASE STATEMENT

---

### 2.2.6 LOOPING CONSTRUCTS

---

#### 2.2.6.1 DO WHILE, REPEAT-UNTIL

---

#### 2.2.6.2 EC-N-TIPLS, EC 1=n TO w BY y

---

#### 2.2.6.3 GENERALIZED LCCE

---

### 2.2.7 DATA

---

### 2.2.7.1 LOCAL VARIABLES

---

### 2.2.7.2 GICAL VARIABLES

---
DO NOT LET THE SIZE AND AMOUNT OF DETAIL OF THIS MANUAL SCARE YOU OFF. IT IS MEANT TO BE DIGESTED IN PIECES, NOT ALL IN ONE SITTING, AS THE COURSE PROGRESSES AND YOUR LEVEL OF UNDERSTANDING AND COMPETENCE INCREASES. BE SURE TO SCAN ALL THE WAY THROUGH THE MANUAL THE FIRST TIME, EVEN THOUGH YOU DON'T UNDERSTAND HALF OF IT, SO THAT YOU WILL AT LEAST KNOW WHERE THINGS ARE WHEN YOU HAVE TO LOOK THEM UP LATER. YOU SHOULD RE-READ THE ENTIRE MANUAL BY MID-SEMESTER, SINCE MANY OF THE THINGS THAT CONFUSED YOU THE FIRST TIME WILL HAVE BEEN COVERED IN CLASS AND WILL MAKE REAL SENSE TO YOU THEN. By mid-semester you will have encountered, through your programming assignments, many of the situations described in this manual.

While we can not (and will not) insist that you strictly observe all the guidelines described here in your first few programs, we will be stricter about enforcing the rules as time goes on. If you find these rules too stringent and restricting, bear with it. You will find, as you get more proficient, that not only will the rules seem much less constraining, but also you will begin to realize when you can safely make exceptions to them. With structured programming, as with all disciplined arts, it is more important that you understand the reasons why things are done than that you memorize a cookbook set of rules for doing them.

Certain concepts explained in this manual may become clearer if you refer to the sample program included at the end of the manual. Any questions that you may have that a look at the sample program does not answer should be referred to a grader. We repeat, however: DO NOT PANIC if you seem to be lost at first. Chances are the concept will be covered in a future lecture.
INTRODUCTION TO THIS MANUAL

This year (as in past years) program documentation will account for roughly 50% of the grade you receive on a program. We thought, then, that it might be nice to give you some idea of how to document a program in such a way that (a) it helps you to design the given program, and (b) it helps us to read the code that you have written, and therefore, most importantly, helps give time for both you and us. The significance of documentation is finally beginning to be realized in the software engineering field; that is, it is being recognized as essential rather than as "something nice if you have time". What follows is a description of a system for documentation in a structured programming environment that has been developed and tested here at Brown in order to make assembly language implementation and documentation easier, less time consuming, and more suitable.¹

Structured programming is not merely a novel way of writing programs. More and more programmers in the "real world,"² are realizing the value of structured, top-down programming to combat constantly increasing costs of software design, implementation, and maintenance. Because the great majority of students in this class have written only small programs as course exercises, it is undoubtedly difficult for you to view program documentation in any perspective other than its obvious relevance to grading in a course. However, programs in the real world are (a) large and complex, (b) subject to re-definition of function over their life spans, and (c) maintained by skilled or unskilled programmers who did not implement the original version. These factors lead to severe problems in program alteration and maintenance. (Well over 50% of the total time, resources, and energy expended on software is for alteration and maintenance.) Hence, complete documentation is absolutely necessary for a program to be of general utility. (If you program in the real world after you leave Brown, you will more than likely not be writing small, one

¹You may wonder how using rules that need 50 pages to explain can save time -- believe us when we say that spending time to do things right in the first place saves a great deal of time and frustration overall, especially in the computer field. So treat this manual as your "stitch in time".
²This is one of those phrases that you will hear often throughout AVD's lectures. Every time you hear it, you know you're in trouble. We now provide you with other such sayings: "Big?", "In the past...", "For tomorrow, all I want you to do is...".

-2-
shet programs, but will participate in group projects in which the skills you learn in AM 100 be necessary).

As part of our software engineering orientation towards the practice and teaching of programming, we have developed a new way to document programs, in which you'll be emphasizing program design, program structure, and program readability -- a major subset of real world documentation. We realize that there may be flaws in the system although we expect to find them ourselves long before you do.3 The essence of the intent of the system we will use is not the learning of its own particulars, but rather the learning of the types of information that must be documented, regardless of the individual scheme employed: clarity, consistency, and completeness are essential. For the purposes of this course, we have defined one illustrative method for documentation that you will all learn to use and abide by, to both your and our profit. Prior to each lecture in which you are introduced to a new programming construct -- i.e., subroutining, I/O, etc. -- we will hand out a sample program illustrating the proper way to document the construct.

The only way that we'll know how successful or unsuccessful this documentation technique, however, is by your criticism. So please feel free to complain to a grader at any time. YOUR COMPLAINTS WILL BE LISTENED TO AND CONSIDERED IN MAKING FURTHER MODIFICATIONS (ADDITIONS OR DELETIONS) TO THE BASIC SYSTEM WHICH IS DESCRIBED IN THE REST OF THIS HANDBOOK. In any case, you are expected to read the rest of this handout and live by its contents until you are told otherwise. The degree to which you understand and follow it will determine a large portion of the grades that you receive on programming assignments.4

3In fact, the system proved sufficiently successful in last year's AM 100 that it has been adapted, with minor modifications, as the Foxboro Company corporate software documentation standard, and has been well received there.

4As an amusing sidelight we might note that a lot of the demos in the SOS manual and most of the "good" programs of five or six years ago would not measure up to our current standards -- the field has progressed a lot, fortunately.
As mentioned earlier, documentation is meant to help you code and debug your programs. As such, we expect you to document as you design and code. Many of you still think of documentation as a separate entity, which takes extra effort and time. We intend to demonstrate that when used correctly, documentation, although it may seem like a bother, actually saves you time and effort. Therefore, one does not wait until the day before the program is due and then add comments to your code and scribble an FLD (a term to be explained later) on the back of a bubble-gum wrapper. We realize that you are basically honest people. We realize that you all mean well and intend to document as you code. We realize that there is no need to check up on you to make sure that you do this. We realize ... the hell we do. We know that you ain't gonna do it unless we make you; so your initial motivation for reading and using this handout is as follows:

WE WILL NOT LOOK AT ANY LISTING -- EVEN IF IT IS UNDERGOING THE INPUT STAGE -- UNLESS IT ACHIEVES THE CONVENTIONS SET FORTH IN THE REST OF THIS HANDOUT.5

5As noted in the Foreword, we cannot hold you strictly to this in the first few assignments, but certainly will thereafter.
1. SUMMARY OF TOP-DOWN DESIGN

This manual is not intended as a first course in top-down design. It is assumed that you are familiar with the concept of top-down design, and this section is inserted here only as a short review. As you remember from Conway and Gries, top-down design, also called step-wise (or successive) refinement, is performed as follows. Sketch out a general, high-level description of your program indicating the overall structure of your program (something like "read in data"; "compute ..."; and "output results" is adequate at this stage). Then take each process that can be further subdivided (like "compute" in the above example) and write a more detailed description for it, expanding and refining the logic still further. Continue the process, going down as many levels as you have to, until your design reaches a level about one level above the actual program statements themselves. At this point, you can start coding.

As you perform this top-down process, use the documentation conventions explained below to help you plan, design, and implement your program.

The documentation you will supply for A.M. 100 programs consists of several parts. Each is explained below in great detail, and with appropriate examples below. An overview of the algorithm and any special (error or boundary) conditions is given in an English Language Description (ELD). Another useful part of documentation is a pictorial, structured flowchart which presents a somewhat more detailed but still macro view of the algorithm and its internal decomposition into modules and their flow of control. You are not required to hand in flowcharts with your other documentation this year, but since you may be using them anyway in the development of your program, why not put them in?

Finally, the listing itself has SGS assembly language with internal documentation consisting of block comments (headers) and "inline" comments in the comments field.

2.1. English Language Description (ELD)

In theory, the English language description of your program tells (in English) what your program does, why it does it, and how it does it. However, you may assume that we have a copy of the assignment; and, therefore, you need only put in your ELD information relevant to your implementation of the problem -- i.e. not the assignment in general. Your ELD, then, should state the following:

2.1.1 User-oriented Information

What your program does from a user's point of view. In most cases a one or two sentence summary of the assignment will do, but if you feel that it is worthwhile to include more material from the assignment definition, feel free to do so.

7Flowcharts are still very common in the real world, so you might as well learn how to do them well.
2.1.2 MODULAR SPECIFICATIONS

This segment of your ELD contains information useful to future maintainers of your program, presented in a standardized outline form called a "modular" specification. There should be one modular specification for each module that comprises your program. In addition to the modular specifications, you should supply an overall description of how all the modules in your system interact. The modular specification consists of a series of topics selected from the set given below, according to which topics are meaningful when applied to your program. All of the following topics should be included in your modular specifications unless they are obviously ludicrous in that context:

2.1.2.1 _NAME_: the name of the module, lest you forget it.

2.1.2.2 _FUNCTION_: a brief description of what your module does and what it is good for (Don't be too detailed; you should have done that when you were giving user information).

2.1.2.3 _CALLED BY_: Primarily useful in large systems for subroutines that get called from a lot of different places. If you change the format of the parameter list for the subroutine, you're going to want to know what routines call it, so you can re-write the call statements.

2.1.2.4 _ROUTINES CALLED_: the same kind of thing as the previous category, for the same reasons.

2.1.2.5 _PARAMETERS_: a listing of all parameters used by this routine, giving the name, type, order, and purpose of each.

2.1.2.6 _CALLING SEQUENCE_: If you must have other than normal linkage conventions (these will be explained later) for passing control to your routine, specify here the exact code that a calling routine must contain to invoke your program. If you use normal SCS conventions, as expected, say so.

---

"Including yourself, supposing that you should want to revive the program sometime later in a new form."

"A module is a section of code that is treated as an entity. As such it is slightly more general than the concept of a subroutine. See section 3."
2.1.2.7. **ENTRY POINTS**: a listing of all points in your routine to which control can be passed by an invoking routine (there should only be one, unless you are doing something really strange).

2.1.2.8. **NORMAL EXITS**: a description of all conditions that prevail when your program exits normally (like return codes, parameter values returned, etc.).

2.1.2.9. **ERROR CONDITIONS**: a description of anything that could possibly go wrong with your program, and (as far as you know) what happens when it does. Be specific and tell everything; if your program barfs on a certain kind of data, it's nice to tell potential users about it so they can avoid supplying that kind of data (and perhaps fix the program).

2.1.2.10. **INPUT AND OUTPUT**: Specify what kind of input your routine expects (if it needs any input), and what kind of output it produces.

2.1.2.11. **GLOBAL ASSUMPTIONS AND SIDE EFFECTS**: Global assumptions are things your program assumes are supplied by calling programs in addition to the standard parameters and linkage conventions (for instance, #12 might point at a common data area used by the whole system). Side effects are changes that your program makes to these global assumptions and variables (and which, therefore, might effect the other programs in the system).

2.1.2.12. **STORAGE UTILIZATION AND DATA**: a description of any major data areas used by your program (particularly if they are complex), and what they are used for. Pictures are very nice. If you have a complex data structure, draw a picture of it and put it in your E.L.E.

2.1.2.13. **ALGORITHM**: Briefly describe the algorithm you used in the program.

2.1.2.14. **DESIGN CONSIDERATIONS**: Whether you wrote the program for general usability, or to solve a specific problem; whether you optimized for execution time, space or something else; what kind of limitations you accepted and built into the design of the program in order to increase efficiency.

---

See Murphy's Law.
2.1.2.15_ COMMENTS: Your personal opinion, or anything else you want to say.

Please note that the above topics are intended as a guideline, not as the last word. Feel free to add to or rearrange the above list, or delete anything that has no relevance (be careful about deletions, however; if your program, for instance, does no I/O, it is usually better to say "I/C -- (none)" than to omit the topic altogether. In any case, if you delete any topic, make sure you supply equivalent information elsewhere in the modular specification. If the above list looks too long and formidable, remember: we're not asking you to type it into every block comment (see below) -- just supply it in the ELD, in any legible form. A relatively easy way to set up a modular specification is by using the "Superdoc" or "Doctor" programs.¹¹

2.1.3 RUN HISTORY

An entry should be present for each time that you ran the program. It should state why you ran it, whether this run found any new errors, whether it showed that errors detected on the previous run(s) had been cleared up, and the cost of the run. For your grader run (and there should only be one) you should also include the number of instructions executed and a rough estimate of the number of executable instructions in your program.

(It is, of course, necessary to set aside a sheet of paper for your run history at the beginning of your work on the program, and enter statistics as soon as you make each run. It can be very disheartening to try to patch together a run history from a random stack of listings the night before the assignment is due -- "Just what was I trying to do here? ...".)

Similarly, we want you to keep track of (and submit as a part of this section) the amount of time you spent on the entire assignment, from initial design, through implementation and debugging. Looking at the figures, we may be able to help you re-think your MODES and improve order to let you spend less time, with better results. In addition, this type of information will enable you to perform better planning and work load estimation prior to

¹¹Information on these programs will be handed out separately.
subsequent programming tasks; estimating the work load of a
task prior to beginning it is always an essential activity.

2.1.4 DESIGN DESCRIPTION

You should include with your ELD some sort of illustration
of how you designed and structured your program. This can
be a flowchart, or it can be a top-down description in a
combination of English and FL/I-like meta-language. In any
case, it should not be as detailed as the structural
descriptions you put in your internal documentation.
Flowcharting is described in detail in section 4.

Keep in mind that an ELD should be written in English and not
blocks of pseudo-FL/I or Assembler code! It should be written
in such a way that a person not familiar with SCS-Assembler
can follow this description of your program. Above all, it
should be brief. For most programs, one or two pages
(preferably typewritten) in addition to the run history and
time history will be sufficient. While it might not seem
possible to fit all the stuff that we're asking for in two
pages, read on. There is a way around this problem by making
use of internal documentation; in fact, as a rule of thumb,
put in the internal documentation whatever parts of the ELD
you'd like to have with you when you're debugging the
listings.
2.2 INTERNAL DOCUMENTATION

The internal documentation is probably the most important part of your program documentation. It is the only part of your documentation that can not get separated from the rest of your program, and it is the most readily available at debugging time. The most important thing you must learn about internal documentation is to TYPE IT IN AT THE SAME TIME YOU TYPE IN YOUR CODE. This point cannot be stressed enough times. The only way you can be sure your documentation will be there when somebody wants it is to be sure it gets put in properly from the first. Do not put it off until the last minute on the assumption that you are going to change your code (or that you are not quite sure just yet what the code is going to look like). You will undoubtedly make several debugging runs in the course of getting your program up; and you will be sitting at a keypunch or terminal editing your program in between runs. During these editing sessions, you can clean up any inaccuracies in the documentation as well as the bugs you found in the code. If the documentation is there to begin with, it is easy to "debug" it in this manner. In any case, it is much easier in the long run than waiting until the last minute and making one huge editing pass through the program inserting documentation the night before the program is due.

You should get in the habit, therefore, of typing a comment on every line of code you enter as you enter it. Do not become too concerned if your column alignment does not match up or you guess wrong about where to put a page object in the listing. You will have other editing sessions in which to correct these problems. The same holds true for block comments. By and large, you have no business writing the code until you have a good idea what your algorithm is. Usually, you do have a pretty good idea of what the code is supposed to do; so put it down. While you're at it, put down your thoughts about what parameters are going to be required and what special error conditions are going to have to be handled. That's it. You have a block comment. It doesn't have to be very long. If you want to add more stuff later, you can go ahead and do it. The point is that you have something there that says the minimum required for a block comment.

We repeat, lest you forget: TYPE IN YOUR DOCUMENTATION AT THE SAME TIME AS YOU TYPE IN YOUR CODE!!

---

12 Like you, three years later.
13 And probably doing a sloppy job because you are in a hurry.
14 Do, however, make an attempt to get these right the first time, so you won't have too much to fix up later.
2.2.1 BLOCK COMMENTS

A block comment is simply a large number of comment lines placed together at the start of a subroutine, program, macro, or other large block of code. Block comments may also be placed at the start of data definitions for complex data structures. Block comments should not be over-used: a program with a block comment every 10 lines of code is almost impossible to read. Block comments also need not be very big. The in-line documentation should manage to explain most of what goes on in your code. Block comments should be short enough so that it is not a chore to type them in, as long as they are big enough to say what they have to say. Do not waste your time typing in huge comments with perfectly aligned boxes around them unless you have the time to waste.

You should have some sort of block comment on every page of code in your program. There should be one big one for the program as a whole; smaller ones for the mainline and each of the subroutines in your program; and little one- or two-liners for continuation pages if a routine is two pages long; and one for each major data area in your program. Theoretically there should be a block comment on each macro you define, but the BOS macro generator is flaky and tends to blow up if there are too many lines (including comments) in a macro, so forget the block comments on macros for the time being. It should go without saying that each of these block comments should be preceded by a page eject, so that your listing is neatly formatted.

Each block comment must contain at minimum a one or two sentence description of what the section it is describing is and does. For subroutines, you should also minimally include a description of the information which must be known both by the subroutine and the calling program, such as parameters (give their names, data types, and usages, and whether their values are passed by the caller to the subroutine or returned from the subroutine to the caller), shared data and registers, abnormal exit conditions, etc. Again, a one-sentence description suffices for each of these categories. For data structures, describe any interrelationships that exist between individual pieces of data in the structure. Block comments for segments of code should also contain information on register usage for that segment, as described in the next section.

---

*If a routine is longer than two pages, it probably should be at least two routines. See section 3.1.
2.2.1.1 REGISTER USAGE

Each block comment that heads a segment of code should contain a section describing the register usage within that segment. Registers are a valuable resource to programmers, since there are never enough of them to hold all the things he would like to keep in them. Therefore it is important to know not only what is in which register at what times, but which registers are unused and may therefore be used if it is suddenly decided that another register is needed. It is also important to flag registers which, although they are not used by the routine in question, are being used by a routine which calls it, and consequently cannot be used without first saving their contents somewhere and restoring them later. The register usage table does all that.

All registers used in the program should be given mnemonic names by which they can be referred to unambiguously, such as R1 for register 1, R2 for register 2, and so on. The reason for this is twofold. First, it makes the program easier to read. The instruction

I R2,0(R1) PICK UP ADD OF FIRST PARM

is much clearer than

I 2,0(1) PICK UP ADD OF FIRST PARM

The second reason is that the mnemonic names for the registers show up in the symbol table listing and (in most assemblers, but unfortunately not SCS) are cross-referenced to the numbers of the statements in which they are used. The EQU assembler pseudo-op is used to assign the mnemonic names. You should make all registers, even when you are not using all of them, just in case you suddenly decide you need another.

Note that the process of setting up names for the registers can be combined with the register usage table for each of your routines (typically the mainline). The register usage table is simply a list containing, for each of the 16 registers, a statement of what it is used for in that particular section of code. If a

---

*SCS only allows you to define a symbol once. For subsequent register usage tables, make the line a comment instead of an EQU statement.*
register is used for two things (like R15 below), put down both usages. If a register is not used, say so. If one or more registers are not used (like R3-R12 below), you may lump them all together on one line. If a register is not to be touched because some other routine is using it, say so and give an indication of who is using it and for what. If a register is used only temporarily to hold the result of a computation which is not to be saved in the register, you may flag it as a work or scratch register. Be brief; the usage statement for each register should rarely exceed one line, and never need take more than two lines. An example of a register usage table is given below:

* R0 WORK REGISTER; NC PERMANENT USAGE
* R1 PARAMETER LIST
* R2 INDEX REG PCB TABLE A
* R3-R12 (NCU USED)
* R13 ***SAVE AREA FCINTER; DC NCU TCUCH***
* R14 RETURN ADDRESS REGISTER
* R15 RETURN CODE REGISTER; WORK REGISTER

2.2.2 INLINE COMMENTS -- GENERAL DESCRIPTION

As we mentioned before, there should be a comment to the right of every assembly-language instruction in your program. These comments are the most important part of your documentation, because assembly language programs are almost undecipherable and unmodifiable without them. Each of these comments should explain the purpose of the statement to its left; and taken together, they should provide a readable commentary explaining how your program does what it does without the reader needing any knowledge of SIS. These comments consist of a mixture of regular English statements and control constructs (such as IF-THEN-ELSE and DO-WHILE) which explain the flow of control in your program. This mixture of ordinary statements and flow-of-control statements is called a "meta-language," because it is similar to high-level programming languages like FL/I, but is used only for documentation purposes rather than for programming.

The distinction between a meta-language and a programming language is an important one. Last year we emphasized the

---

17 If you are combining register usage tables with EQU’s (as in the mainline), then obviously you cannot put several registers on one line.

---

1974
similarities between our meta-language and PL/I, in the
hopes that it would make the transition from PL/I to
assembly language easier, and would help people understand
how the structured programming they learned in A.M. 51 was
just as valid in assembler. As a result of this, some
people got the impression that they were to write a PL/I
program down the comments column of their listings and
"hand-compile" it into assembly language. Then they got
into trouble when they had written a statement such as "R3
= INDEX(SUBSTR(CARE,1),"");" and then tried to translate
it to assembler. You have to write a whole set of
subroutines to do something like that in assembler. People
also defined a whole bunch of inscrutable variable names,
like NLPRT and CLDWP and N_WD_LN and then tried to use
them in PL/I-like comments like "N_WD_LN=NNLEN-MEW_V-5;".
Such comments are less readable than undocumented assembly
language. Even though the meta-language we ask you to use
for internal documentation looks somewhat like PL/I or any
other high-level programming language, it is not a
programming language, but a descriptive one. Individual
comments are written in ENGLISH and are carefully aligned
in neat, indented 19 columns to illustrate the logical
structure of the program.

2.2.3 STATEMENT COMMENTS

There is not too much to say about comments on ordinary
instructions in the program. However, one or two points
must be emphasized. First, they must be written in ENGLISH,
as was stated in the preceding section. Do not use only
PL/I statements or FORTRAN statements in the comments
column: if these languages were self-sufficient as
documentation, you wouldn’t have to document programs
written in them. Do not invent obscure and abbreviated
variable-names to represent values in your registers or
anywhere else. It is better to say "INCREMENT LCCE COUNTER
FOR <LCENMHO>" than "I = I + 1"; and both will fit into
the 30-40 columns you should have to fit them in.20 The
only variable names that should appear at all in your
comments are the labels you define in your code.

Secondly, make sure each line of comments you type
describes the assembly language statement it appears
alongside. This may seem self-evident, but it has happened

19To show nesting of loops, etc.
20And if, perchance, the first shouldn’t fit, you can drop
off the reference to the loop name without causing much
ambiguity.
with reasonable frequency that someone has gotten carried away with writing an assembly language program down the left and a meta-language program down the right and has needed two lines of meta-language somewhere to describe one line of assembly language code, and then gone and put the continuation of the comment line in the comments field of the next statement; and continued thus, with the assembly side being one statement ahead of the comments the rest of the way down. When you need to use an extra line to explain a particularly tricky assembly-language statement, insert a line with an asterisk in column 1 (a comment card) and continue your comment on that according to the rules below "for continuing comments. Do not simply repeat what the assembly language instruction says in your comment. For instance, if the instruction is "SLL R5,1", a comment which says "SHIFT R5 LEFT ONE BIT" adds no new information. In this case, your comment should say something about why the shift was done. For instance, if you have a word of flags in R5 and were sitting in a loop shifting one by one into the sign position in order to test them, you could put a comment like "SHIFT NEXT FLAG BIT TO SIGN POSITION" on the line. The important thing is not that the comment merely reflect what the statement does, but that it explains what the statement does in a larger context.

Sometimes a comment has to be continued on more than one line; or a comment may apply to more than one statement, as in the case of a long computation. In such cases, the comment line may be continued on the next line by placing a period (.)21 on the next line in the column directly beneath the start of the original comment. The text of the continuation is then typed following the continuation character. For example, the following piece of code assumes R6 contains the address of an entry in a table called TABLE and computes the index from the start of the table. Each entry in the table is two words long. Presumably you have prior comments explaining what R5, R6, and R15 hold, since these comments do not.

```
L      R5,P6
INDEX OF ENTRY = (ADDRESS OF
SXAI  R15,TABLE(R15). ENTRY - ADDRESS OF
S      R5,R15
      TABLE)
SPA    R5,1
      DIVIDED BY 2 (ENTRY SIZE)
AXAI   R5,1
      + 1
```

The examples showing the flow-of-control constructs later on also show the use of the comment continuation principle.

21 Or some other character you like to use as a continuation marker, such as a hyphen.