DOCUMENTATION FOR THE REAL WORLD:
Simple, Timely, and Effective!

Originally presented at the 1989 Interex *Computing Management Symposium*
Nashville, TN March 8-10, 1989

A. Marvin McInnis
5250 W. 94th Terrace, #114
Prairie Village, KS 66207

INTRODUCTION

Documentation has long been the stepchild of software development. It is commonly postponed until the final phases of a project, and all too often it is postponed forever. Technical documentation is often inaccurate, incomplete, and/or incomprehensible, and user manuals, when written, are frequently unusable for their intended purpose. In extreme cases, it becomes impossible to determine from the documentation whether software is functioning correctly or not!

This paper presents a system of documentation, incorporating ideas from a number of sources, which has proven to be very effective in producing useful, accurate, and timely documentation. The essential ideas are that appropriate documentation is an important aspect of a successful project, and that the documentation should be an integral part of the project and not a separate activity. Managers may be surprised to discover that an integrated documentation plan can actually decrease the development time of a project. Other benefits are clearer completion criteria, improved software quality control, increased user acceptance, and reduced maintenance.

WHAT IS "GOOD" DOCUMENTATION?

The ultimate purpose of software documentation is to facilitate understanding of what the software is supposed to do and how it is supposed to do it. We can usually recognize good documentation when we see it, but despite all that has been written, it is still difficult to define what determines its "goodness." Quite simply, creating good documentation remains an elusive art, but we can identify several characteristics of good documentation:

- Appropriateness: Good documentation is appropriate to the complexity of a software system and its intended use. Clearly, the documentation requirement for a programmer's toolkit utility is different than for production software or software for non technical users.

- Clarity and Accuracy: Good documentation is both understandable and technically accurate. This implies that documentation must address several different levels of detail; the documentation needed by a non-technical user is very different from that needed by a maintenance programmer.

- Timeliness: Good documentation is timely. User manuals or technical documentation which do not accurately reflect the current state of the software are of severely limited value. Even worse are user manuals which appear only after the users have developed their own superstitious, trial and error understanding of the system.

- Careful preparation: Good documentation is carefully prepared and well written. Manuals which are poorly written or otherwise difficult to use won't be used very often, which is not much better than no manuals at all.

- Consistency: Good documentation is both logically and stylistically consistent. Consistency is essential to allow a both technical and non technical users to correlate the contents of documents at different levels of detail. Logical consistency simply means that all documents should reflect the same concepts and ideas, while stylistic consistency implies that these concepts and ideas should be expressed in a uniform way from one document to the next.

- Flexibility: Finally, good documentation should be flexible and dynamic, rather than static. Documentation must easily be changed, always remaining "in sync" with the software as the system is developed, released, maintained, and enhanced.

HIDDEN BENEFITS OF GOOD DOCUMENTATION

We all know the reasons why projects should be documented, but I would like to suggest several less obvious benefits of adopting an integrated documentation program.

A good documentation plan can involve end users early in a project, and the result can be software which is better suited to users' actual needs and more readily accepted by them. Good documentation, produced early in the development cycle, can minimize problems due to misunderstandings among users, managers, and the technical staff.

Good documentation will almost always result in a shorter development cycle, better software quality, faster testing, and reduced debugging. This is important to you as managers: a good documentation program will reduce software development time even when compared with the case of no documentation at all, and it will probably reduce the total development cost as well.

In his excellent presentation "The Seven Secrets of Successful Software Project Management" at the 1988 Interex Computing Management Symposium, Vaughn Mantor made the interesting observation that a common problem of software development is that no ever knows when the project is finished! Good documentation can help resolve this dilemma: when the software is demonstrated to perform in accordance with the documentation, the initial development project is complete. This means that not only must the data be handled correctly, in accordance with the functional specifications, but the user interface also must function correctly and in accordance with the user manuals.

Good documentation can also benefit both the purchaser and the vendor of a commercial software product:

For the purchaser, the quality of the documentation will usually be a reliable indicator of software quality. Good documentation can define the acceptance criteria for a software product, and it can reduce the purchaser's dependence on the vendor's support services for the life of the product.

For the vendor, good documentation can be a very effective marketing tool, will increase customer satisfaction with the product, and can reduce the cost of customer support. The explicit use of documentation as acceptance criteria for a software product is as desirable for the vendor as for the purchaser: performance in accordance with the documentation can obviate most disputes over product acceptance.

BUILDING BLOCKS OF GOOD DOCUMENTATION

Although the details will vary depending on the needs of your organization, there are five fundamental components of any good documentation system:

1) Programming and documentation standards

2) Functional specifications

3) User manuals

4) Technical documentation

5) Program source code

Note that there are two generic varieties of user manuals, the reference manual and the tutorial, and most software will require both. The primary purpose of the reference manual is to provide accurate and detailed information on using the software for experienced users. Conversely, the tutorial should be a self-study introduction to the software, and it should assume that the user has little or no experience with the subject matter. Since the user tutorial and reference manuals are intended for completely different purposes, it is a good idea not to combine them into a single manual.

FIVE STEPS TO EFFECTIVE DOCUMENTATION

Just having each of the components in place doesn't constitute a good documentation plan unless they are coordinated with each other in a systematic way. Below are five steps, developed over a number of years, which provide a simple and effective documentation system. This system works. It has proven successful for a number of my clients, and I recommend that you try it:

1) Develop documentation standards and make a commitment to documentation.

2) Write the user reference manual first, and make it a part of the project functional specifications

3) Develop the technical documentation in a three-level hierarchy

4) Make the documentation, not the software itself, the key items of project control

5) Validate the documentation as well as the software during testing

Note that the basis of this system is the "top down" development model: the documentation at any given level should be well defined before any work is started at a lower (more detailed) level. For example, rather than being left until last, drafting the user reference manual should be one of the first activities in the software development cycle. If you are already using the "top down" model for your software development, this system of documentation will be especially easy to adopt.

DEVELOPING DOCUMENTATION STANDARDS

A documentation system just won't work without some kind of standards. And while most organizations have established programming standards, few have any standards at all (the worst kind of standards) for documentation.

Don't misunderstand. I'm not suggesting that you adopt an elaborate and rigid set of documentation standards, but rather that you begin now to develop an evolutionary set of standards appropriate to your organization and your software projects. The very existence of standards will be beneficial, because they establish the importance of documentation in your organization and confirm the commitment to it

Documentation standards can serve several purposes, but the key ideas here are that you should adopt standards of some kind and that those standards should be appropriate to your organization. At the very minimum, documentation standards should:

- State the goals of the documentation plan
- Define the "deliverable" documents for different categories of software, the relationships among these documents, and when they will be required
- Establish minimum standards for content of each required document
- Provide guidelines for style and content (including examples) which will promote clarity and consistency

If your organization doesn't have any documentation standards at the present time, start developing them now! Begin with very simple, easy to follow, standards. Then try to keep them simple, adding to them only when necessary.

MAKING A COMMITMENT TO DOCUMENTATION

A documentation plan will surely fail if there is not a management commitment, at all levels of your organization, to its success. As a manager, you will play an important part in this success.

The employees you manage must understand that good documentation is as important to the success of the project as good design and good coding. And they must trust that you will continue to support the importance of their documentation efforts, even when things get rough.

The commitment to documentation will occasionally require courage on your part: the courage to say "No, the project will not be released until it is finished," adding documentation to your completion criteria.

In short, in order to be successful you need to have a clear idea of why documentation is important to your organization, be willing to support it before your employees, and be willing to defend it before your managers.

WITE THE USER REFERENCE MANUAL FIRST!

This is the most important idea that I have to share with you here: WRITE THE USER REFERENCE MANUAL FIRST!

The user reference manual is the one document defining a system that can be understood by everyone involved in the project: users, managers, analysts, and programmers alike. For this reason, a working draft of the user reference manual should be written as early in the project as possible, and this draft manual should be a part of the project functional specifications.

The user reference manual should be an exact representation of the way the system will appear to users. In purely functional terms, users should be able to view the system entirely in the context of the reference manual; the system internals should be "black box" entities totally transparent to users.

Besides being an important system design document, the user reference manual also provides a good control mechanism for system design changes which will affect the users. If technical changes will result in changes to the user reference manual, which implies that user approval will be required, the manual functions to allow all parties to understand the affect of the changes on their ÿrequirements. Conversely, when changes are requested by the users, they can submit their requests in the context of changes to the manual.

The format and content of the user reference manual will vary, depending on the organization and the specific project, but I recommend that it include at least three separate sections:

1) An index based on the user's view of the system (i.e. an index organized by user job function rather than by software module function) which defines user access capabilities available and refers to the appropriate pages in the other two sections of the manual

2) Screen formats and report layouts

3) Procedure descriptions, including a summary of user interaction and expected results, for each screen and report

The last two sections can be combined into one if desired, but I prefer to keep them separate. The

procedure descriptions are usually much easier to work with when the screen layouts can be viewed side by side with them.

See figures 1 through 3 for examples taken from a draft user reference manual which was written for an actual system functional specification.

WRITING GOOD USER MANUALS

No doubt about it, writing good user manuals is an art. But it is an art which can be learned by most people. Although a thorough treatment of how to write good user manuals is far beyond the scope of this presentation, I would like to recommend to you what is absolutely the best reference I have seen on the subject, and one of the simplest. It was originally written for internal use at the Jet Propulsion Laboratory (NASA), but is now available from The Society of Technical Communication:

> "Guidelines for Preparing Software User Documentation"
> by Diane F. Miller
>
> available from:
>
> The Society of Technical Communication
> 815 15th Street NW
> Washington, DC 20005
> Attn: Mary Johnson (202) 737-0035
>
> ISBN book number 0-914548-57-3
> STC book number STC-133-88

The book is only about 80 pages, but it is packed with information and ideas, and it also provides an excellent example of the techniques it advocates. Below are the book's major recommendations for writing good user manuals (used by permission of the author):

- Establish a clear purpose for the manual
- Analyze your audience
- Gather complete information
- Make sure information is accurate
- Organize for easy reference
- Design the (physical) package to fit
- Format for quick comprehension
- Use visuals and examples liberally

- Use standard conventions and terminology
- Use a clear and concise style
- Edit ruthlessly

I believe that you can vastly improve the quality and usefulness of your user manuals by adopting these simple principles. Try them and see.

PREPARING TECHNICAL DOCUMENTATION

The technical documentation for a project should be organized in a three-level hierarchy consisting of a technical overview, a logical synopsis of each software module, and a detailed prototype of each module (rendered in pseudocode).

At the top level, the primary purpose of the technical overview is to restate the functional specifications in terms of implementation, dividing the project into discrete tasks and defining each of them. There will be only one technical overview per project.

The middle level should consist of a logical synopsis of each software module identified in the technical overview. The function of the synopsis is to outline its subject module at a level of detail much greater than in the overview, with particular emphasis on logical organization, data characteristics, algorithms, and processing procedures. There may be many synopses for a project, but each there will be only one synopsis for each software module.

The bottom level of technical documentation should comprise a detailed prototype of each software module. Each synopsis will expand into one and only one prototype, and the prototype should represent an accurate expression of the constructs embodied in the actual source code for the module. In most circumstances, some standardized pseudocode will be used for the prototype, although your organization may adopt a different standard.

Explanatory text may be included at any level of the documentation, when needed for clarification. Try to set explanatory text off from body text by indenting or by using a contrasting type face, and keep it as short as practical; if explanatory text takes more than a paragraph, that whole section of the document probably needs to be reorganized.

Flowcharts, diagrams, and other illustrations may also be used to clarify the text of a document. I find diagrams particularly useful in illustrating error handling sequences, unusual algorithms, or inter-module communications protocols, but please don't waste time on flowcharts which add little or no additional understanding to the body of the document!

In some circumstances, the technical overview may be combined with the functional specifications. Similarly, the module synopses may be combined with the related prototypes, but the key idea is that the three documentation levels constitute a logical hierarchy which is consistent from one level to the next, in a process very much like outlining.

THE TECHNICAL OVERVIEW

While the primary purpose of the functional specifications is to state what the project is supposed to accomplish, the purpose of the technical overview is to provide a high-level statement of how those goals will be accomplished. At the very minimum, the technical overview should:

- divide the project into specific modules and tasks
- state the purpose of each module and the relationships among modules
- define basic data structures and data flow
- define standards for inter-module communication
- provide some estimate of resource requirements
- fully characterize any external constraints

As stated previously, for simple projects the technical overview can often be combined with the functional specifications, but remember that the combined document still must effectively serve two different purposes.

THE MODULE SYNOPSIS

The module synopsis is a logical outline, at a non-detailed level, for each software module defined in the technical overview, and its primary purpose is to describe the specific procedures the module employs. The emphasis at this level should be on clearly describing the functionality of each module without getting into coding details.

Note that each synopsis relates back to a single section of the technical overview, and relates forward to a single module prototype document.

A module synopsis should include (as appropriate):

- identification of the project
- the module name
- a descriptive title
- the module revision level
- a short description of the purpose of the module (abstracted directly from the technical overview)
- a processing outline of the module, with emphasis on logical organization, control and data flow, basic algorithms used, etc.

- explanatory text and/or illustrations

Figure 4 contains an example of a portion of a synopsis excerpted from an actual project.

THE MODULE PROTOTYPE

The module prototype is a detailed expansion of the synopsis into a logically complete "template" for the software source code, but still expressed in a narrative form (pseudocode) rather than in program language statements. Since the primary purpose of the prototype is to function as a "template" for the related source code, it must implement the concepts expressed in the synopsis in a detailed, logical form which is readily understandable by programmers, analysts, and other technical personnel.

Note that for each module identified in the technical overview there is a single module synopsis and a single prototype. Maintaining logical consistency between a synopsis and its related prototype is extremely important, and the best way to accomplish this is to expand each line of the processing outline in the synopsis into a stand-alone paragraph heading in the prototype.

Figure 5 contains an excerpt of pseudocode corresponding to the module synopsis example in figure 4. Note how the documentation structure ÿmakes it very easy to relate the synopsis to ÿthe corresponding pseudocode.

Each prototype should include (as appropriate):

- the module name and descriptive title
- input and output specifications
- files accessed
- data base records accessed
- "include" files required
- project-specific procedures called (subroutines, functions, etc.)
- system-independent procedures called
- system-dependent procedures called
- external modules (programs) called
- pseudocode

The first draft of a prototype will usually reveal most logical errors present in the synopsis, and they are much easier (and faster) to correct at this point than at the source code level. And in addition to helping to identify design errors early in a project, the formal process of logical

expansion from technical overview to synopsis to prototype also encourages good programming practices in a way that is not oppressive to designers, analysts, or programmers.

THE SOURCE CODE

The hierarchical style of technical documentation should extend down to the source code itself: each line of pseudocode in the module prototype should appear as a stand-alone comment line in the source code.

An extremely simple and powerful technique is to use a copy of the module prototype document as a "skeleton" for the program source file. Editing each line of pseudocode into a stand-alone comment line quickly yields a "pre-commented" source code file which conforms exactly with the documentation. I recommend that you also add some unique "signature" characters to each of these comment lines; I'll show you why later.

Figure 6 is an example of FORTRAN source code corresponding to the synopsis in figure 4 and the the prototype pseudocode in figure 5. Note that each comment line generated from the pseudocode includes "==>" as signature characters, while other comment lines do not.

Each line of pseudocode should become the heading for no more than one paragraph of source code; if you have more than one paragraph of source code per line of pseudocode, it is a clear indication that the pseudocode is not detailed enough.

CONTROLLING THE PROJECT

If you follow my plan, the documentation will be virtually complete before the coding has gotten very far along. You can now use the documentation to control the project: system design reviews, phase checks, and change requests all can be handled entirely in the context of the documentation!

Progress reports, change requests, and problem reports should all be discussed in the context of the highest level of documentation which will be affected. Clearly, the higher the level of documentation affected, the greater will be the effects on the whole system design; for example, an error at the synopsis level will almost always be more severe and pervasive than an error in the prototype pseudocode, and will require the involvement of higher level project managers.

Progress reports should always be presented in terms of the appropriate level of documentation. A common problem in software project meetings is that valuable management is wasted in discussing the programming details of a software module; the real problem is that without documentation there is nothing to discuss but the code itself. With a good documentation plan, the only reasons to review the actual code should be for audit purposes, to verify conformance with the prototype pseudocode, or to review compliance with programming standards.

Similarly, change requests should be submitted in the context of changes to the documentation. User requests for changes should be submitted as requests for changes to the user reference manual, and user involvement should be mandatory for any other change requests which will affect the user manuals.

Problems, or bug reports, should also be submitted in terms of the highest level of documentation affected. At the coding level, normally an error will require changes to the code alone (a

programming error) or to the prototype pseudocode (usually an algorithmic error); if changes are required at higher documentation levels, they may point to serious errors in the system design.

Of course, all of the documentation must be updated whenever a change is adopted, for whatever reason. Just as important, revised documentation should be distributed to all interested parties as soon as it is accepted. It is extremely important that the revision level of the documentation currently distributed should always match the revision level of the underlying software.

TESTING AND QUALITY CONTROL

The test phase of a project is also the time to validate the documentation, both for conformance with the actual software implementation and for logical consistency among documents.

The user reference manual should be a primary component of the project test program. Each and every statement in the reference manual should be tested and verified. This may seem obvious, but often it is not done, leading to user manuals which confuse and frustrate users. If a discrepancy is discovered, either the manual must be changed (with user involvement!) or the software must be changed to conform with the manual.

The test phase is also a good time to write the user tutorial manual if one is required. If possible, the primary author of the tutorial should be a user who participates in the software testing. It is important that each and every instruction in the tutorial be tested and verified, even more than in the reference manual: an important secondary function of the tutorial is to instill a new user with confidence in the reliability of the software and the documentation.

A simple and powerful first step in validating the technical documentation for a module is to generate an abstract of the module source code. The abstract should consist either of all stand-alone comment lines appearing in the module or just those comment lines containing the special "signature" characters discussed earlier. The UNIX tool SED(1) can be used to generate the abstract, or you can use some other CASE tool or even write a simple abstracting program yourself.

Compare the abstract of the source code with the prototype pseudocode; they should be nearly identical. Figure 7 shows part of an abstract of the source code module which was used for figure 6. Comparing figure 7 with figure 5, it is easy to verify that the software logic faithfully follows the prototype pseudocode.

If desired, the source code abstract can be "collapsed" further and compared with the module synopsis documentation. Logical consistency from the abstract to the prototype to the synopsis is what we are looking for here. Discrepancies between the prototype and the synopsis usually indicate management problems with project control.

Upon acceptance and release of the software, there should be absolute consistency between each software module, the documents comprising the technical documentation, and the user manuals. You should accept nothing less.

A SHORT CASE STUDY

In 1987 and 1988 I worked on a fairly large project in which different modules of the software were supplied by the client, by a prime contractor, and by my company which was a subcontractor.

This project employed many of the documentation strategies described here, and it was quite successful.

As a subcontractor, we were provided with a well designed set of functional specifications and a technical overview which included screen layouts and a narrative description of user interaction with the system. There were no user manuals, but the technical overview contained enough information for software development.

No more than three meetings with the client were required per module, with most requiring only the first and last two:

1) a design review prior to coding, at which the client reviewed and approved the module synopsis and the prototype

2) if required, a review meeting for significant design changes, with re-approval of the revised technical documentation

3) the module delivery and acceptance, which involved discussion of the final documentation and demonstration of the software itself

It is significant that all of the management of our work on the project was conducted in the context of the documentation, and the only discussions about coding dealt with client standards issues.

Because of a good documentation plan and good management, our work on the project was on time and within budget. And I think the documentation plan allowed us to do some pretty good work: more than six months after acceptance, the client has reported exactly zero bugs in our software! In short, I wish all projects were this successful.

ACKNOWLEDGEMENTS

I hope that you have found some useful ideas for project documentation in my presentation. I would like to acknowledge the clients and professional colleagues whose ideas helped me in the evolutionary development of the concepts presented here:

- Paul Chatfield, Hewlett-Packard Company
- Jim Cummins, Allied Signal
- Brian Horn, South Africa Broadcasting Corporation
- Bill Jackson, Allied Signal
- Diane Miller, Jet Propulsion Laboratory

Correspondent capability allows the authorized user access to the following functions:

Task Assignment - The ability to enter an "unknown" story after entering the required assignment data relating to the story

Library Search - The user is able to enter a new story, edit a story, and update the story status. The story entered is placed in the COPY INPUT file.

Program Function	Sub Function	Menu	Key	Screen Page	Procedure Page
Task Assign.		Main	A	G-8	H-6
Library Search		Main	S	G-14	H-7
	Display Story	Search	f1	None	H-9
	Print Story	Search	f2	None	H-9
	Change Story	Search	f3	G-10	H-9
	Edit Script	Search	f4	None	H-12

--------- Figure 1 - Sample User Capability Matrix ---------

CORRESPONDENT ENTRY

```
(30)        * SABC *                    18:34    23/08/84

 REFERENCE   AIR DATE                                DIV    STATUS
[___AUTO__]  [CCCCCC]                                [*]     [CC]

TITLE    [CCCCCCCCCCCCC]  EVENT DATE [CCCCCC]  SOURCE [****]
LOCATION [************]  EVENT TIME [****]

CATEGORY [**]       REPORTER [AUTO]       CAMERAMAN A [****]

COMMENTS [************************************************]
         [************************************************]
         [************************************************]
         [************************************************]
```

NOTE: 1. Fields indicated by an asterisk (*) are those into which data may be entered.

2. Fields indicated by a 'C' are compulsory entry fields and if data is not already present in these fields it must be entered.

Figure 2 - Sample Screen Layout

2.1 TASK ASSIGNMENT

The news gathering process begins with the input editor completing part of a pre-formatted form. It is used by the reporters, writers, and output editors. As it passes through each phase of broadcast preparation, final information about the aired story may be added.

This form will become the permanent part of the library record after the broadcast. The library file includes the slug with related film, slide, and tape information. Full text archiving is present in the system, and the full script is appended to the form.

The Task Assignment function is used by the Input Editors to create new assignments for reporters. It is also used by Correspondents when entering new stories not previously assigned by an Input Editor.

To enter the Task Assignment function from the main menu display:

Press **A <RETURN>**

The Task Assignment screen (see page G-8) will be displayed.

To create a new assignment you must enter five required items:

1) Air date (this may be a tentative date)
2) Event date (normally today's date)
3) Story title
4) Reporter
5) Status

You may optionally enter any available information into the other fields on the display. (See section G for a description of each data field and its usage.) You should enter all information known about the story at this time, even though it may change later. Note that the Reference Number and Input Editor fields will be entered automatically by the system.

---------- Figure 3 - Sample Procedure Description ---------

Module Name: ADMON (CMM Module 05.0)

Title: CMM Audit Verification

Revision date: May, 1988 (original)

Description:

ADMON is the program in the CMM Data Base Executive Software System which implements audit control of Part Programs throughout their development cycle. ADMON provides a user audit and update interface to the CMM part program data base (PPDB) and the QDM data base.

Synopsis:

A) Initialize program (entry from CMM Main Menu)

1) Get true terminal LU and attach ADMON to it
2) Get system data from data block program
3) Get parameters (e.g., printer LUs) for this terminal LU from data block program
4) Open data base (user IDs) for shared read
5) Open data base (part programs) for shared read
6) Initialize FORMS subsystem

B) Main menu (module 05.1)

1) Display ADMON main menu (screen AD001)
2) Wait for user input
3) Branch according to menu selection:
f1 - help
f5 - return to CMM Main Menu (exit)
f8 - next screen (continue)
(continue on f8):
4) Get user ID and password from screen
5) Validate password
6) Get user audit capability from data base
7) Go to Section C (select for audit)

C) Select part program for audit display (module 05.2)

1) Display ADMON menu (screen AD002)
2) Wait for user input
3) Get part part number, suffix, and equipment

---------- Figure 4 - Excerpt of Module Synopsis ----------

```
------------------------------------------------------------

Section --> (B) <-- Main menu (module 05.01)

Error handling:

All IMAGE and FORMS errors are fatal.  The 3rd successive
timeout on the CRT terminal will in all cases result in an
error being logged, followed by an immediate shutdown and
exit.

Processing:

B.01 ==> Display ADMON main menu screen (screen AD001)

Deactivate previous screen (if any)
Activate screen AD001
Display screen AD001
Zero timeout counter

B.02 ==> Wait for user input

Wait for user input
If CRT timeout:
   Increment timeout counter
   If 3rd successive timeout:
      Display timeout message line
      Go to --> G <-- (Shutdown)
   else (1st or 2nd timeout):
      Display timeout message line
      Return to --> (B.02) <-- (Wait for user input)
   endif
else:
   Zero timeout counter
endif

B.03 ==> Branch according to menu selection

If function key f1 'Help':
   Display help screen
   Return to --> B.02 <-- (Wait for user input)

else if function key f5 'Main Menu':
   Set good completion status
   Go to --> G <-- (Shutdown)

else if function key f8 'Next Screen':

---------- Figure 5 - Excerpt of Module Prototype ----------
```

```
------------------------------------------------------------

**==>*****************************************************
**==> Main menu (module 05.01)                             *
**==>*****************************************************

C ==> Deactivate previous screen (if any)

 1000 call F_DEACTIVATEFORM (F_ERROR)
      if (F_ERROR .eq. 0) then        ! OK
         continue
      else if (F_ERROR .eq. 13) then  ! no form was active
         continue
      else                            ! FORMS error
         call CMM_LOG_ERROR (5100, F_ERROR, BLANK)
      endif

C ==> Activate screen AD001

      call F_ACTIVATEFORM (AD001, F_INFO, FMP_ERROR,
     &   F_ERROR)
      if (F_ERROR .eq. 0) then        ! OK
         continue
      else if (F_ERROR .eq. -1) then  ! FMP error
         ERROR_STRING = AD001
         call CMM_LOG_ERROR (5102, FMP_ERROR, FMP)
         go to 9000 ! shutdown
      else  ! (F_ERROR .ne. 0)        ! FORMS error
         call CMM_LOG_ERROR (5104, F_ERROR, FORMS)
         go to 9000 ! shutdown
      endif

C        Move terminal LU to screen

      call F_PUTSTRING (SLU, LUT_C, 3, F_ERROR)

C ==> Display screen AD001

 1100 call F_SHOWFORM (F_ERROR)
      if (F_ERROR .eq. 0) then        ! OK
         continue
      else  ! (F_ERROR .ne. 0)        ! FORMS error
         call CMM_LOG_ERROR (5106, F_ERROR, FORMS)
         go to 9000 ! shutdown
      endif

C        Move blank employee ID to screen
```

--------- Figure 6 - Excerpt of FORTRAN Source Code --------

```
------------------------------------------------------------

**==>*****************************************************
**==> Main menu (module 05.01)                             *
**==>*****************************************************
C ==> Deactivate previous screen (if any)
C ==> Activate screen AD001
C     Move terminal LU to screen
C ==> Display screen AD001
C     Move blank employee ID to screen
C     Display data fields
C ==> Zero timeout counter
C ==> Wait for user input
C ==> If CRT timeout:
C ==>    Increment timeout counter
C ==>    If 3rd successive timeout:
C ==>       Display timeout message and shut down
C ==>    else (1st or 2nd timeout):
C ==>       Display timeout message wait for input
C ==> else (user input):
C ==>    Zero timeout counter and continue
C*-----------------------------------------------------*C
C* Screen AD001 - f1: Help                            *C
C*-----------------------------------------------------*C
C ==> If function key f1 'Help':
C ==>    Display help screen and wait for input
C*-----------------------------------------------------*C
C* Screen AD001 - f5: Return to main menu             *C
C*-----------------------------------------------------*C
C ==> If function key f5 'Main Menu':
C ==>    Shut down
C*-----------------------------------------------------*C
C* Screen AD001 - f8: Next screen                     *C
C*-----------------------------------------------------*C
C ==> If function key f8 'Next Screen':
C ==>    Get employee ID number and password from screen
C ==>    If employee ID or password is blank:
C ==>       Display error message and go wait for input
C ==>    Read employee ID record from data base
C ==>    If no ID record found:
C ==>       Display error message and go wait for input
C ==>    Decrypt password from data base record
C ==>    If passwords don't match:
C ==>       Display error message and go wait for input
C ==>    Expand user capability bit map into separate flags
C ==>    go to ADMON MENU section (screen AD001A)
C*-----------------------------------------------------*C

------------ Figure 7 - Abstract of Source Code ------------
```