Title: VAX-11 Software Engineering Manual -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SEPRR3.RNO

PDM #: not used

Date: 19-Feb-77

Superseded Specs: none

Author(s): F. Bernaby, P. Conklin, S. Gault, T. Hastings, P. Marks, R. Murray, I. Nassi, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: This manual presents the VAX-11 programming conventions and software engineering practices as developed for, and adopted by the Central Engineering Group. These conventions and practices are standard within Central Engineering; we hope that they be used by other corporate groups as well. Designed to be the "programmer's helper", the manual contains the coding conventions as well as practical data of technical, procedural, administrative and conceptual nature that would be useful to the software engineer.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	19-Feb-77

Rev 2 to Rev 3:

1. Convert to standard RUNOFF manual chapter format.

2. Remove unwritten sections.

3. Move introductory note to preface.

4. Remove request for review from preface.

5. Note relationship to BLISS conventions.

6. Split chapter 6 into 6 and 7. Add chapters 8-11, especially BLISS. Add the BLISS transportability guidelines. Add Chapter 15 for diagnostics.

[End of SEPRR3.RNO]

VAX - 11

SOFTWARE ENGINEERING MANUAL

19 FEBRUARY 1977

Revision 3

DO NOT DUPLICATE

For additional copies, contact:

Ike Nassi
ML 21-4/E20

Digital Equipment Corporation, Maynard, Massachusetts

Revision 1, April, 1976
Revision 2, June, 1976
Revision 3, February, 1977

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

This draft standard does not describe any program or product which is currently available from Digital Equipment Corporation. Nor does Digital Equipment Corporation commit to implement this standard in any program or product. Digital Equipment Corporation makes no commitment that this document accurately describes any product it might ever make.

Digital Equipment Corporation's software is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

CDP
COMPUTER LAB
COMSYST
COMTEX
DDT
DEC
DECCOMM
DECUS
DECsystem-10
DECsystem-20
DECtape
DIBOL
DIGITAL
DNC
EDGRIN
EDUSYSTEM
FLIP CHIP
FOCAL
GLC-8
IDAC
IDACS
INDAC
KA10
KI10
LAB-8
LAB-K
MASSBUS
OMNIBUS
OS/8
PDP
PHA
PS/8
QUICKPOINT
RAD-8
RSTS
RSX
RT-11
RTM
SABR
TYPESET-8
TYPESET-10
TYPESET-11
UNIBUS

PREFACE

Over the years, much ado has been made about coding standards and conventions. Everyone believed that conventions are good, so long as they are not the other guy's conventions! Committees were formed, and reformed, and left to die for lack of consensus. We repeatedly refused to follow conventions that we deemed "imperfect" and consequently we followed none at all.

A great deal of this has been foolish nit-picking on the part of our vast multitude of entrepreneurs. The time has come to stop the foolishness and to recognize the reasons for which code uniformity is mandated.

Standards, conventions and uniform practices all aid us in producing reasonably professional, maintainable products of consistent quality. Any individual can always have a private opinion as to what is "good", or "right", or "efficient" or "aesthetic". Any collection of individuals invariably comes up with as many divergent opinions on the subject as there are individuals. We should all be sufficiently mature and sufficiently professional to be willing to compromise with both our egos and our fellow peers; to compromise just enough to accept objectively a set of reasonable conventions that will establish the uniformity and consistency of all of our software products.

The Methodology group has compiled the conventions and practices presented in this manual. They apply to all VAX-11 programming. They are based on existing PDP-11 coding practices. This manual was reviewed by the Coding Conventions Committee consisting of Peter Conklin, Dave Cutler, Roger Gourd, Steve Poulsen and Mike Spier. These conventions have been broadened to the BLISS environment by review with Ron Brender, Rich Grove, and Dave Tolman. Transportability issues have been addressed in concert with Peter Marks and Ike Nassi.

We want these conventions to be adopted willingly, not forced upon people through arbitrary managerial edict. This is best accomplished by having you formulate to yourself exactly WHY you find some convention to be objectionable; then try and propose --to yourself-- an alternate one, and reflect on whether or not the new one is really that much superior, and why. All that we ask of you is to convince yourself that these conventions are no less reasonable than any other set of conventions. Then, we hope, you will be willing to show sufficient professional maturity to adopt and follow these conventions.

This document is the result of integrating and reorganizing the BLISS Software Engineering Manual and the VAX Assembler Software Engineering Manual published during the summer of 1976. New chapters have been incorporated, covering transportability, naming conventions, and external interface specifications. We solicit constructive criticism and recommendations for enhancement. In particular, the last chapter contains a list of topics we would like to address in future editions. Please feel free to contribute toward these topics. This is completelv a home grown document. If you feel this is a desirable way to proceed, you should feel a responsibility to review this carefully and to contribute material you feel appropriate. The value of the document depends directly on the quality and applicability of the submitted material.

CONTENTS

Page

[End of Prefix]

Title: VAX-11 Software Engineering Introduction -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE1R3.RNO

PDM #: not used

Date: 23-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: The introduction gives a chapter by chapter overview of the manual and how it is organized.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	23-Feb-77

Rev 2 to Rev 3:

1. Change to be the general contents and guide to the chapters.

2. Add Chapter 7.

3. Add Chapter 8.

4. Add purpose of manual.

5. Split chapter 6 into 6 through 11.

6. Add chapters 14 and 15.

7. Collect loose ends as last chapter (# 99).

[End of SE1R3.RNO]

CHAPTER 1

INTRODUCTION

23-Feb-77 -- Rev 3

This manual is concerned with software engineering practices in the VAX-11 environment. It does not discuss or define the differences between VAX-11 and other environments. Designed to be the "programmer's helper", the manual contains the coding conventions as well as practical data of technical, procedural, administrative and conceptual nature that would be useful to the software engineer.

This manual has two purposes:

- to provide the Software Engineer with information not normally found in language reference manuals such as usage notes and symbol construction rules.
- to present recommended standards, conventions, and practices such as commenting, formatting, and documentation.

Conventions, standards and practices can assure good, professional, maintainable products of consistent quality. They need not encroach on the programmer's "right" to be creative in his or her expression of a program.

Chapter 1 is the introduction and gives a guide to the manual's organization. It includes a chapter by chapter overview.

Chapter 2 tells how to use this manual. It tells how to find the exact information needed. It also gives the notations used in the manual.

Chapter 3 is the methodological policy statements. These are the policies which lead to the specifics of the format. They also outline the basic structure of programs into modules. The policy statements include the goals to be attained by following them. These policies include the choice of language, the layout of the source text, the separation into modules, and the sharing of code.

Chapter 4 is a program structure overview. It lists the source module's textual elements, and gives examples of the parts of the program. This pulls together in one place the details documented later in chapter 6.

Chapter 5 gives the standard module template files and the instructions for using them. The standard template contains all of the standard boilerplate as a convenience to save excessive retyping.

Chapter 6 details the commenting conventions. These are consistent across all source languages. The entries are arranged alphabetically for ease of reference. There is extensive cross-referencing to aid retrieval. For each item, it gives the background and the rules, and then gives templates and examples.

Chapters 7 through 11 give usage and formatting conventions for each of our programming languages. The languages covered are assembler, BASIC, BLISS, COBOL, and Fortran. Although there is occasional redundancy between these chapters, we felt it better to minimize retrieval difficulty at the expense of some duplication. The chapters are layed out in the same style as Chapter 6. When a topic deserves more than a page to describe, an outline is given here and a cross reference is made to a fuller presentation in some other chapter.

Chapter 12 is the naming conventions. These include the formation of symbols reserved to Digital and the list of facility prefixes.

Chapter 13 gives details on forming external and interface documentation. In particular, it includes details on the notation for specifying procedure arguments.

Chapter 14 contains guidelines for the transportation of BLISS programs across architectures.

Chapter 15 contains additional information and guidelines for writing diagnostics programs.

The last chapter is a collection of loose ends and future sections.

The appendices give full sample programs written to this standard.

[End of Chapter 1]

Title: VAX-11 Software Engineering How to Use -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE2R3.RNO

PDM #: not used

Date: 26-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 2 gives a guide to the use of the manual and gives its notations. It suggests ways of looking up information in it.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	26-Feb-77

Rev 2 to Rev 3:

1. Replace usage cross reference notation.

2. Note split of commenting and usage chapters.

[End of SE2R3.RNO]

CHAPTER 2

HOW TO USE THIS MANUAL

26-Feb-77 -- Rev 3

This manual assumes familiarity with the VAX-11 languages. Its purpose is to serve as a guide to the precise use to which certain language features may be put.

The introduction (chapter 1) indicates the chapters of the manual, explaining what each chapter contains. The sts table of contents lists individual sections within the chapters. The index is organized by keywords (e.g., COMMENT, ROUTINE, STATEMENT, etc.)

Suppose that you were told that your program needs better comments. You should typically look up the concept under "C" in the chapter on commenting. Similarly, if you were told that your useage or formatting of some source statement was poor, you could look it up under the statement's name in the chapter on formatting and usage for your language.

This will enable you immediately to retrieve the information required, and have the exact amount of information that is pertinent to your immediate needs. You may then get additional information about the keyworded item in the other chapters. The important point is that such additional information is not confused with the information needed for some specific reason. The manual is deliberately not organized for front- to back-cover sequential reading.

Keyworded data is cross referenced. The rules pertaining to keyword "A" may require knowledge or use of keywords "B" and "C".

- o Knowledge of "B" and "C": a "SEE ALSO" pointer indicates the related item(s) which you should also understand.

- o Use of "B" or "C": the first occurrence of "B" and of "C" within "A" is prefixed with the word "see" serving as a reference pointer to indicate the possible need to consult those keywords in turn.

There may be variants of a single keyworded concept. For example LABEL and LOCAL LABEL. In this case, the keywords are ordered by the main concept (e.g., LABEL), and any variant is to be retrieved by suffixing that keyword with the qualifying key word. We use the colon ":" as a qualification delimiter within the manual (e.g., LABEL: LOCAL).

| Finally, whenever this manual is reissued, all changes relative to the immediately preceding version of the manual will be indicated by means of a left margin change bar, as illustrated to the left of this entire paragraph.

[End of Chapter 2]

Title: VAX-11 Software Engineering Policy -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE3R3.RNO

PDM #: not used

Date: 26-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 3 gives the methodological policy statements. These include the choice of language, the layout of the source text, the separation into modules, and the sharing of code.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	26-Feb-77

Rev 2 to Rev 3:

1. Remove reference to page boundaries.

2. Allow code in application languages.

3. Document reasons for structure and for transportability.

4. Limit interface data types to call standard.

5. Remove references to self-initializing.

[End of SE3R3.RNO]

CHAPTER 3

METHODOLOGICAL POLICY

26-Feb-77 -- Rev 3

| 1. All system programs for the VAX-11 family are written in an application language or one of the two official system implementation languages:

- o The VAX-11 Macro Assembler, or
- o BLISS-32

Of these, BLISS is the default choice for a language. BLISS is intended to replace as much assembly code as possible. The assembler will be used as a system implementation language only for:

- o Hardware dependent routines, such as interrupt handlers or I/O drivers, where extreme machine dependency coupled with high performance requirements rule out the use of BLISS.
- o Cases where functionality is needed that is not supplied by BLISS; for example, routines which are to be invoked in a non-standard way.
- o Routines which cannot be written in BLISS because of compilation difficulty (as distinct from functional impossibility, or undesirability). This category includes all routines which would have been coded in BLISS had there been available a BLISS compiler that supports the required technicalities (e.g., special relocation or addressing features). All these routines are, in principle, candidates for future recoding in BLISS, conditions permitting.

2. All code will be written uniformly, according to these conventions, in order to:

   o Make system code meaningfully readable. If source code is not properly structured, organized, and indented according to these conventions, you have obscured the algorithm from the reader. The code should be structured into blocks with a limited amount of branching. This allows a graphical reflection of the control flow. If the code is unstructured, you have lost the ability to understand and modify it.

   o Enable all programmers to read, understand and be able to modify one another's code, regardless of source language. Note that the documenting conventions are identical across all our languages.

   o Enable field support personnel (both software specialists and hardware engineers) to read and understand VAX-11 system code. To lower field support costs by eliminating, as much as possible, the need for software specialists who are knowledgeable of certain routines only, and to further the software specialist's ability to master any system code.

   o Make our software well documented: make programs both readable and comprehensible by being able to extract technical documentation from the source code itself. Facilitate the work of technical writers by providing them with uniform, well documented source code.

   o Reduce the bug rate and enhance the quality and stability of our software products. Maintain the product's initial high quality throughout its lifetime: through cycles of bug fixes, modifications and functional evolution.

3. All major bodies of code, or distinct logical sub-systems (with the exception of speed/size sensitive executive or diagnostic modules), will be coded as independent routines using the standard call/return interface, to:

   o Encourage and facilitate the use of BLISS in non-critical sections of system software, and to

   o Encourage the future recoding of assembly language routines in BLISS, conditions permitting.

   o Enhance the ability to transport non-assembly language code.

   o Limit the interface data types to those specified as part of the calling standard.

4. All user-level system products (lanquage processors, utilities, library subroutines, etc.) should be desiqned and implemented so that they may be transportable between systems and/or family architectures. Keep in mind that:

   - Transportability is a major qoal to which Central Engineerinq is firmly committed.

   - Transportability has to be desiqned carefully into the product, and carefully realized by followinq the transportability guidelines.

   - All machine-dependent features are to be avoided as a rule. If necessary, they should be localized to a clearly-identifiable, non-transportable module.

5. The sharinq of code is encouraged as much as possible. Whenever possible, use a library service routine instead of codinq your own version of that same function. If such a library routine does not yet exist, code one that is of qeneral nature, and submit it to the library.

6. All proqrams are to be written modularly, in small self-contained modules that are maintained as individual source files. These modules will be assembled separately. The object code files will be linked to form the larger software product. Modularity will benefit us by:

   - Enhancing quality: each module can be tested and debugged separately; small modules are more easily controllable than large bulky proqrams.

   - Isolating functionality: it becomes easier to custom tailor a system through selective linkinq of exactly those modules that are needed.

   - Enhancing modifiability: the modification of a qiven module will be less likely to have an undesirable side effect on some other module's functionality.

   - Workinq set control: the ability to rearranqe the linkinq order of modules is a most powerful tool in optimizinq proqram behavior within a paged runtime environment.

7. All modules (with the possible exception of certain core executive or diagnostic programs) are to be written as pure, non-self modifying and well localized code.

- o Self initializing: With the exception of system startup or bootstrap code, all routines should be self initializing. If they depend on an initial value of some permanent allocation (OWN) variable, initialize that variable dynamically rather than relying on compile time or link time value settings.

- o Well localized: VAX-11 is a virtual memory machine. Any piece of code may --whether originally intended to, or not-- possibly run in a demand paging environment. You should make the greatest efforts possible to design and structure your code in such a way that the locality of reference is kept to a minimum. Don't promiscuously branch over a large absolute address span. Don't make reference to widely (and wildly) fragmented database elements within a single sequence of instructions, and especially within the scope of a tight loop.

[End of Chapter 3]

Title: VAX-11 Software Engineering Program Structure -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE4R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 4 overviews and then details the layout of a module. It includes examples of the module and routine prefaces.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Remove the VERSION NUMBER statement.
2. Explain why routine owns are discouraged.
3. Update to use template in the example.
4. Define calling sequence vs. input and output parameters.
5. Add references to $FORMAL, etc., macros.
6. Change CONFIGURATION to ENVIRONMENT.
7. Combine abbreviated and detailed edit history.
8. Add weak and validation section.
9. Add BLISS to show similarity.
10. Add critical algorithms to functional description.
11. Use NONE for inapplicable sections, do not delete them.
12. Title and ident are first two lines.
13. Legal notices are fully capitalized.
14. Edits have initials if several editors per version.
15. Ident examples include edit.
16. BLISS module head includes other module switches.
17. BLISS structure defs are together.
18. Add blank line after legal notices.

[End of SE4R3.RNO]

CHAPTER 4

PROGRAM STRUCTURE

28-Feb-77 -- Rev 3

Programs are written in modules. The module is the source text that is assembled or compiled as a unit. Each module can be coded in any language. The program structure and commenting conventions are consistent across all languages to allow the reader to learn one pattern independent of the writer's choice of language. Also, for reader ease, every section and subsection must appear in its standard position. If a section or comment is not applicable, enter the word NONE as a separate line. This is done to make the reader's job as simple and clear as possible. Each module exists as a separate source text file, and is structured as follows:

4.1 THE MODULE PREFACE

It provides the necessary documentation to explain the module's functionality, use and history. It consists of the following items in the exact given order. All items must be included.

- A title statement specifying the module's name. The title is a symbol of up to 15 characters in length. This statement has a comment indicating the module's functionality. The title statement, together with its comment, are reproduced as page headers in the listing. The title statement is always the first line of the file.

- An IDENT statement indicating the module's current version number. The ident statement is always the second line of the file.

- The standard DEC legal notices fully capitalized for emphasis.

- A FACILITY statement. A module may be a dedicated part of a larger linked facility, or part of several facilities, or a general purpose library function. This statement identifies the larger whole of which the module is part.

- A short functional description of the module (a documenting comment) including the design basis for any critical algorithms. If the module requires an extensive functional description, then this item is an abstract of the description, and is identified as such by the keyword ABSTRACT. The extensive functional description will then be provided on the following page.

- ENVIRONMENT statement. Give any special environmental assumptions such as access modes, OTS, etc. If the module's assembly is governed by a system wide configuration file, then state the file(s)'s name(s). Otherwise if the module has special conditional assembly parameters, then specify very explicitly what they are and what values they assume under all given conditions.

- The author and date on which the module was coded.

- The detailed current edit history. This item specifies the versions, the modifier, and the last date of each version. This item also lists the specific changes made between base levels (during production) or releases, providing a short functional description of each problem and its solution, as well as appropriate reference information such as SPR number(s), etc. The comments include the full name of the person responsible for each version. If several people modify the module, the initials of the others appear in each edit line.

4.2 THE MODULE'S DECLARATIVE PART

It contains:

- For BLISS, specification of the table of contents.

- Specification of INCLUDE files or library definitions.

- Definition of local macros.

- Declaration of local equated symbols

- Declaration of own storage allocations.

- Specification of externals. For assembly language, only WEAK or VALIDATION externals need be listed.

4.3 THE MODULE'S ACTUAL CODE

This is in the form of zero or more ROUTINE(s). The module may have no routines in it (i.e., no executable code) if it is a DATA SEGMENT MODULE. Each routine consists of the following sequence of items:

4.3.1 The ROUTINE PREFACE

- A routine statement specifying the routine's name. This statement has a comment indicating the routine's functionality. The routine statement, together with its comment, are reproduced as page headers in the listing.
- A detailed functional description of the routine.
- A list of the routine's calling sequence, input and output parameters.
- A list of the implicit inputs and outputs, and functional side effects, if any, of the routine's code.

4.3.2 The Routine's Declarative Part

- Specification of local INCLUDE file(s), if appropriate. Normally, such use is not recommended.
- Declaration of local (stack frame resident) variables.
- Declaration of optional equated symbols, own storage allocation variables and macros, all of which are local to this routine. In general, use of these local items is not recommended unless it adds significant clarity. Usually, these are better declared at the module level.

4.3.3 The Routine's Code

- For assembly language, the routine's entry point(s).
- The routine's body.
- The routine's return instruction.

4.4 MODULE TERMINATION

An end module statement terminates the module.

4.5 ANNOTATED SAMPLE LAYOUTS

The above are explained in detail in the commenting and formatting chapters of this manual. In the following sections a sample layout of the module format is presented. Samples are given for both assembler and BLISS coding to show the similarity.

The following notations are used to designate source listing formatting:

- <new page> indicates an inserted form feed "CTRL/L" character or an assembler .PAGE directive, to force the listing onto a new page.

- <separator> indicates either several (normally=4) <skip>s or a <new page>. A <separator> is indicated wherever it would be desirable to force a new page, if the present page is sufficiently full. If the last section only marginally fills the present page, and the following item of text would remain on the page, then they can both appear on the same page separated by several blank lines.

- <skip> indicates a blank line.

- <space> indicates a single blank character.

- <tab> indicates a horizontal tab character.

4.6 SAMPLE LAYOUT OF THE MODULE PREFACE

4.6.1 Example Of The Assembler Module Preface

```
        .TITLE  EXAMPLE - <terse functional description>
        .IDENT  /03-05/

;
; COPYRIGHT (C) 1977
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
;
; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
; REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;

;++                                 <this is a DOCUMENTING COMMENT>
; FACILITY: General Library
;
; FUNCTIONAL DESCRIPTION: (or ABSTRACT:)
;
;       A short 3-6 line functional description of the module.
;       If an extensive functional description is called for,
;       then this should be a short abstract.
;
; ENVIRONMENT: User Mode with OTS
;
; AUTHOR: Charlie Brown, CREATION DATE: 4-Jul-76
;
; MODIFIED BY:
;
;       Lucy vanPest, 17-Aug-76: VERSION 02
; 01    -  Program Crashes if Disk Error
; 02    -  SPR #4711: reads incorrect block after error.
;
;       Snoopy Beagle Brown, 19-Dec-76: VERSION 03
; 03    -  SPR #5391: reads blocks backward if 50 hertz.
; 04    -  Power fail recovery not reliable
; 05    -  (LVP) SPR #5432: recover if ECC recoverable.
;--                                 <end of DOCUMENTING COMMENT>
<new page>
```

4.6.2 Example Of The BLISS Module Preface

```
MODULE EXAMPLE ( ! <terse functional description>
        IDENT='03-05'
        <other module switches>
        ) =

!
! COPYRIGHT (C) 1977
! DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
!
! THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
! COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
! ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
! MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
! EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
! TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
! REMAIN IN DEC.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
! CORPORATION.
!
! DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
!

!++                                  <this is a DOCUMENTING COMMENT>
! FACILITY: General Library
!
! FUNCTIONAL DESCRIPTION: (or ABSTRACT:)
!
!       A short 5-6 line functional description of the module.
!       If an extensive functional description is called for,
!       then this should be a short abstract.
!
! ENVIRONMENT: User Mode with OTS
!
! AUTHOR: Charlie Brown, CREATION DATE: 4-Jul-76
!
! MODIFIED BY:
!
!       Lucy vanPest, 17-Aug-76: VERSION 02
! 01    -  Program Crashes if Disk Error
! 02    -  SPR #4711: reads incorrect block after error.
!
!       Snoopy Beagle Brown, 19-Dec-76: VERSION 03
! 03    -  SPR #5391: reads blocks backward if 50 hertz.
! 04    -  Power fail recovery not reliable
! 05    -  (LVP) SPR #5432: recover if ECC recoverable.
!--                                  <end of DOCUMENTING COMMENT>
<new page>
```

4.7 SAMPLE LAYOUT OF THE MODULE DECLARATIONS

4.7.1 Example Of The Assembler Module Declarations

```
        .SBTTL  DECLARATIONS
;
; INCLUDE FILES:
;

        <library INCLUDE files and library macros which define:
             MACROs, assembly parameters, systemwide equated
             symbols, table definitions>

;
; MACROS:
;

        <local macro definitions>

;
; EQUATED SYMBOLS:
;

        <equated symbol definitions>

;
; OWN STORAGE:
;

        <declaration of permanent storage allocations>
        <also local storage structures, etc.>
        <if many structures, give each a heading>
        <see $OWN and structure macros>

;
; WEAK AND VALIDATION DECLARATIONS:
;

        <only include section if any declared>

<new page>
```

4.7.2 Example Of The BLISS Module Declarations

```
!
! TABLE OF CONTENTS:
!

        <forward routine declarations in order with
             a summary description of each>

!
! INCLUDE FILES:
!

        <library REQUIRE files and library macros which define:
             MACROs, assembly parameters, systemwide equated
             symbols, table definitions>

!
! MACROS:
!

        <local macro definitions other than structure definitions>

!
! EQUATED SYMBOLS:
!

        <LITERAL and BIND declarations>
        <when a group of structure, macro, and literal declarations
         define a structure they should be grouped together here>

!
! OWN STORAGE:
!

        <declaration of permanent storage allocations>
        <also local storage structures, etc.>
        <if many structures, give each a heading>

!
! EXTERNAL REFERENCES:
!

        <externals with short description>

<new page>
```

4.8 SAMPLE LAYOUT OF THE ROUTINE PREFACE

4.8.1 Example Of The Assembler Routine Preface

```
        .SBTTL  EXAMPLE - <short one-line description>
;++                           <this is a DOCUMENTING COMMENT>
; FUNCTIONAL DESCRIPTION:
;
;       <detailed functional description of the routine>
;
; CALLING SEQUENCE:
;
;       <instruction for calling this routine>
;       <include AP-list if applicable>
;       <see $FORMAL macro>
;
; INPUT PARAMETERS:
;
;       <list of explicit input parameters other than AP-list>
;       <typically registers or stacked arguments>
;
; IMPLICIT INPUTS:
;
;       <list of inputs from global or own storage>
;
; OUTPUT PARAMETERS:
;
;       <list of explicit output parameters other than AP-list>
;       <typically registers or stacked results>
;
; IMPLICIT OUTPUTS:
;
;       <list of outputs in global or own storage>
;
; COMPLETION CODES:
;
;       <list of R0 completion codes>
;       <if standard function, change heading to FUNCTION VALUE>
;       <if the hardware condition codes are set,
;               change the heading to CONDITION CODES>
;
; SIDE EFFECTS:
;
;       <list of functional side effects including environmental changes>
;       <exclude implicit outputs of global or own storage>
;       <list all SIGNALs generated if any>
;--                            <end of DOCUMENTING COMMENT>
<separator>
```

4.8.2 Example Of The BLISS Routine Preface

```
ROUTINE EXAMPLE (arguments) =    !<short one-line description>
!++                               <this is a DOCUMENTING COMMENT>
! FUNCTIONAL DESCRIPTION:
!
!       <detailed functional description of the routine>
!
! FORMAL PARAMETERS:
!
!       <list formal parameters and give documentation of them>
!
! IMPLICIT INPUTS:
!
!       <list of inputs from global or own storage>
!
! IMPLICIT OUTPUTS:
!
!       <list of outputs in global or own storage>
!
! COMPLETION CODES:
!
!       <list of function value completion codes>
!       <if standard function, change heading to FUNCTION VALUE>
!
! SIDE EFFECTS:
!
!       <list of functional side effects including environmental changes>
;       <exclude implicit outputs of global or own storage>
!       <list all SIGNALs generated if any>
!--                               <end of DOCUMENTING COMMENT>
<separator>
```

[End of Chapter 4]

Title: VAX-11 Software Engineering Template -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE5R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: MARS template by R. Gourd

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 5 presents the standard template files. It also includes step by step instructions for editing them to form a module in standard format.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Add instructions from Gourd memo RSG028 Rev 2.

2. Update to latest MODULE.MAR punctuation.

3. Abstract is in one space, not one tab.

4. Add instructions for editting modifications.

5. Add configuration to the environment section.

6. Add instructions to include $FORMAL macro.

7. Add weak/validation section.

8. Add instructions for .ENTRY.

9. Document using intials in maintenance history.

10. Max source line should be 80 columns.

11. Add BLISS template.

12. Add blank after legal notices; add blank after abstract.

[End of SE5R3.RNO]

CHAPTER 5

TEMPLATE

28-Feb-77 -- Rev 3

Included here are instructions for commencing a module of coding, a copy of the template file which is the basis of a new module, and instructions for filling in the template.

5.1 MAKING A NEW ASSEMBLY LANGUAGE MODULE

When you commence the writing of a program in VAX-11 assembler language, you should work from a copy of the template file MODULE.MAR, which contains the proper formatting for assembler programs.

\ MODULE.MAR is on the PDP-11 MIAS system under [202,1]. To commence creation of your own module, simply type

```
PIP filename.MAR=DB0:[202,1]MODULE.MAR
```

where "filename" is your designated file name (nine characters or less). \

MODULE.MAR is normally available under the VAX-11 system by copying from the system assembler library directory

```
$COPY MARS_LIB:MODULE.MAR filename.MAR
```

where "filename" is your designated file name (nine characters or less).

Once your copy of the module template exists you must fill in and/or alter certain information prior to writing code.

A copy of MODULE.MAR is shown at the end of this section. The line numbers in the left margin are for reference in this tutorial; they are not part of the file. Refer to Chapter 4, the Program Structure Overview, for an overview of the various sections. Refer to Chapters 6 and 7 for details on each section. Refer to the Appendix for a sample program.

line 001 Replace "TEMPLATE" with your module name and put a terse (half line) description to the right of the hyphen (-).

line 002 Enter the version number between the two slashes.

line 025 After the colon, enter the name of the facility within which the module resides (e.g., system library, math library, etc.).

line 028 After this line, enter a terse (3 to 6 lines) summary of the functionality of the module, starting each successive line with ";<tab>".

line 030 After the colon, describe the environment within which this module (code) will run, e.g., at what access mode, whether it has interrupts disabled, interrupt level, etc. Include any conditional assembly instructions here.

line 032 Following the first colon and <space>, enter your name; follow the second colon and <space> with the creation date of the module.

line 036 As versions are released, copy this line after the replicated line 037. After the <tab> which is before the comma enter the modifier's name. After the space after the comma enter the modification date. Update this date everytime the file is editted. At the end of the line enter the version number.

line 037 As edits are made after first release, copy this line changing the edit number. At the end of this line describe the edit. If the individual making the change is different from the one responsible for this version, then put the changer's initials in parentheses at the start of the description of each edit.

lines 043 047 051 055 Make appropriate entries in each defined section (reference Chapter 4 if you don't understand the section titles named on template lines 041, 045, 049, and 053).

line 055 Follow this line with a section of weak and validation declarations if any.

line 056 Replace "TEMPLATE_EXAMPLE" with your routine's name and follow the hyphen with a half line description.

line 059 Enter a sufficient description of the function(s) of this routine, starting each successive line with ";<tab>".

line 063 If this module is "called", replace "NONE" with the calling sequence (AP-list). Otherwise give the instruction for invoking this routine.

lines 067
071 075
079 083
087

When applicable, replace "NONE" with the information required by the section titles named on template lines numberred 065, 069, 073, 077, 081, and 085.

line 091 If the routine is CALLed, define its formals by including a $FORMAL macro here.

line 093 Replace "TEMP_EXAMPLE" with your. routine's name

line 092 Preceed the semi-colon with the entry mask or first instruction and adjust the comment appropriately. Or merge lines 093 and 094 into a .ENTRY statement.

line 095 Commence the body of your routine/module, commenting appropriately thoughout. Keep source lines to 80 columns maximum.

line 096 Replace "TEMP_XMPL_EXIT" with your routine's exit location label.

line 097 Replace and/or delete the inappropriate return instruction from this and the succeeding line.

LISTING OF MODULE.MAR

```
        .TITLE  TEMPLATE -
        .IDENT  / /

;
; COPYRIGHT (C) 1977
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
;
; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
; REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;

;++
; FACILITY:
;
; ABSTRACT:
;
;
; ENVIRONMENT:
;
; AUTHOR:       , CREATION DATE:
;
; MODIFIED BY:
;
;       , : VERSION
; 01    -
;--
```

```
  <page>
                .SBTTL  DECLARATIONS
        ;
        ; INCLUDE FILES:
        ;

        ;
        ; MACROS:
        ;

        ;
        ; EQUATED SYMBOLS:
        ;

        ;
        ; OWN STORAGE:
        ;

```

```
<page>
        .SBTTL  TEMPLATE_EXAMPLE -
;++
; FUNCTIONAL DESCRIPTION:
;
;
; CALLING SEQUENCE:
;
;       NONE
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       NONE
;
;--



TEMP_EXAMPLE:
        ; ENTRY POINT (OR MASK)

TEMP_XMPL_EXIT:
        RET
        RSB

        .END
```

5.2 MAKING A NEW BASIC LANGUAGE MODULE

Details to be supplied.

5.3 MAKING A NEW BLISS LANGUAGE MODULE

When you commence the writing of a program in BLISS, you should work from a copy of the template file MODULE.BLI, which contains the proper formatting for BLISS programs.

\ MODULE.BLI is on the IPC PDP-10 System-F under BLI:. To commence creation of your own module, simply type

```
          COPY filename.BLI=BLI:MODULE.BLI
```

where "filename is your designated file name (six characters or less). If the module is not transportable use output file type .B32 to indicate this. \

MODULE.BLI is normally available under the VAX-11 system by copying from the system BLISS directory

```
          $COPY BLISS_LIB:MODULE.BLI filename.BLI
```

where "filename" is your designated file name (nine characters or less). If the module is not transportable use output file type .B32 to indicate this.

Once your copy of the module template exists you must fill in and alter certain information prior to writing code.

A copy of MODULE.BLI is shown at the end of this section. The line numbers in the left margin are for reference in this tutuorial; they are not part of the file. Refer to Chapter 4, the Program Structure Overview, for an overview of the various sections. Refer to Chapters 6 and 9 for details on each section. Refer to the Appendix for a sample program.

line 001 Replace "TEMPLATE" with your module name and put a terse (half line) description to the right of the exclamation (!)

line 002 Enter the version number between the two apostrophes. Add any other module switches one per line after line 002.

line 027 After the colon, enter the name of the facility within which the module resides (e.g., system library, math library, etc.).

line 030 After this line, enter a terse (3 to 6 lines) summary of the functionality of the module, starting each successive line with "!<tab>".

line 032 After the colon, describe the environment within which this module (code) will run, e.g., at what access mode, whether it has interrupts disabled, interrupt level, etc. Include any conditional compilation instructions here.

line 034 Following the first colon and <space>, enter your name; follow the second colon and <space> with the creation date of the module.

line 038 As versions are released, copy this line after the replicated line 039. After the <tab> which is before the comma enter the modifier's name. After the space after the comma enter the modification date. Update this date everytime the file is editted. At the end of the line enter the version number.

line 039 As edits are made after first release, copy this line changing the edit number. At the end of this line describe the edit. If the individual making the change is different from the one responsible for this version, then put the changer's initials in parentheses at the start of the description of each edit.

line 046 Enter all routine names defined in this module one per line. Terminate each except the last with a comma. Follow each with a short summary comment (half line). Keep the routines in the order of occurence in the module. Include any routine attributes needed by BLISS.

lines 051 055 059 063 Make appropriate entries in each defined section (reference Chapter 4 if you don't understand the section titles named on template lines 049, 053, 057, and 061).

line 069 Enter all external references made by your routine here one per line. Terminate each except the last with a comma. Include any necessary attributes. Follow each with a terse summary comment of its purpose (half line).

line 070 Replace "TEMP_EXAMPLE ()" with your routine's name and its formal parameter list. Put a terse description of the routine to the right of the exclamation (!). If the above two edits will not fit on this line, keep the comment on this line and place the formal parameter list on the next line. If your routine returns a value, delete the ":NOVALUE" and enter the routine value(s) in the section entitled "ROUTINE VALUE:" (line 091).

line 074 Enter a sufficient description of the function(s) of this routine, starting each successive line with "!<tab>".

line 078 If this module has parameters, replace "NONE" with the list of all parameters in order one per line. For each give a complete description including the passing mechanism in formal notation.

lines 082 086 091 095 When applicable, replace "NONE" with the information required by the section titles named on template lines numberred 080, 084, 088, 089, and 093. Delete whichever of lines 088 and 089 is not applicable.

line 102 List the routine's locals one per line. Follow each with its attributes and a descriptive comment.

line 103 Commence the body of your routine/module, commenting appropriately thoughout. Keep source lines to 80 columns maximum.

line 104 Replace "TEMP_EXAMPLE" with your routine's name.

LISTING OF MODULE.BLI

```
MODULE TEMPLATE (        !
                IDENT = ' '
                ) =
BEGIN

!
! COPYRIGHT (C) 1977
! DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
!
! THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
! COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
! ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
! MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
! EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
! TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
! REMAIN IN DEC.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
! CORPORATION.
!
! DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
!

!++
! FACILITY:
!
! ABSTRACT:
!
!
! ENVIRONMENT:
!
! AUTHOR:       , CREATION DATE:
!
! MODIFIED BY:
!
!       , : VERSION
! 01    -
!--
```

```
<page>
!
! TABLE OF CONTENTS:
!

FORWARD ROUTINE
        ;                ! 

!
! INCLUDE FILES:
!

!
! MACROS:
!

!
! EQUATED SYMBOLS:
!

!
! OWN STORAGE:
!

!
! EXTERNAL REFERENCES:
!

EXTERNAL ROUTINE
        ;                ! 
```

```
<page>
ROUTINE TEMP_EXAMPLE () :NOVALUE =      !

!++
! FUNCTIONAL DESCRIPTION:
!
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

    BEGIN

    LOCAL
        ;                !

    END;                            !End of TEMP EXAMPLE
<page>
END                                 !End of module
ELUDOM
```

5.4 MAKING A NEW COBOL LANGUAGE MODULE

Details to be supplied.

5.5 MAKING A NEW FORTRAN LANGUAGE MODULE

Details to be supplied.

[End of Chapter 5]

Title: VAX-11 Assembler Software Eng. Commenting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE6R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 6 gives each piece of the commenting conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Change column numbers to start with 1 instead of 0.

2. Change CF to FP.

3. Use lowercase English for character names instead of bracketting them.

4. Change example comments to not waste a leading space.

5. Add sections for Author, Calling Sequence, CASE instructions, Comment: group, Completion codes, Condition Handler, Conditional Assembly, Environment statement, Facility statement, Function Value, Functional Description, Inplicit Inputs and Outputs, Interlocked Instructions, Libraries, Listing Control, $LOCAL Macro, Macros, $OWN Macro, Parameters: Input and Output, Program, Queue Instructions, Routine: Order, Side Effects, Signals, String Instructions, Structures, Synchronization: Process, UNWIND, .VALIDATE Declaration, .WEAK Declaration. Add many cross references and sections which are there only to cross reference to another section.

6. Combine abbreviated and detailed history.

7. Add ;++ format.

8. Change symbol definition mechanism from Spier to STARLET.

9. State when dual names might be justified.

10. Clarify when to renumber local labels.

11. Add Call by descriptor.

12. Clarify when <separator> can be four blank lines.

13. Change terminology:
    Routine to Procedure (where appropriate)
    Subroutine to Routine: non-standard
    Definition to Declaration
    Copyright to Legal Notices

14. Move symbol naming rules to Chapter 7. Add references to chapter 8.

15. Change examples to use template formats and text.

16. Put configuration Statement in Environment statement.

17. Document that entry mask must include registers on non-standard subroutines.

18. Change to use .ENTRY.

19. Change [VALUE] to Chapter 8 notation.

20. Move Comment to Comment: Line.

21. Document maintenance numbers. Don't reset them on release.

22. Omit license paragraph for unlicensed software.

23. Add that labels should be meaningful.

24. Give rules for file name.

25. Never use lower case in symbols. Freely use underline. Choose names to suggest attributes.

26. Add that functional description should include critical algorithms.

27. Note that arg list is read only.

28. Include typical .PSECT attributes.

29. Fill in the external symbols section.

30. Split into chapters 6-7.

31. Note that as matter of taste can put a space after the comment delimiter.

32. Make completely language independent.

33. Move contents of completion codes here from naming conventions.

34. Emphasize that legal notices must be on the first page.

35. Emphasize that no keyword is to be omitted; instead use NONE.

36. Emphasize that both the blank comment lines and the blank lines are mandatory.

37. Add support letters to version standard.

38. Add that numbers and letters are not skipped in version, update, or patch.

39. Add update to version standard.

40. Add examples to version standard.

41. Remove attention grabber outdenting.

42. Allow for edit numbers to be facility wide if appropriate.

43. Completion codes have <2:0> of symbol non-zero. Test with CMPV.

44. Add customer version numbers.

45. Move procedure to chapter 7.

46. Move maintenance comments to end of line.

[End of SE6R3.RNO]

CHAPTER 6

COMMENTING CONVENTIONS

28-Feb-77 -- Rev 3

This chapter contains detailed information on commenting conventions. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

The notation <comment delimiter> is used to represent the comment delimiter of the source language. For example, this is a semicolon (";") in assembly language and an exclamation mark ("!") in BLISS and Fortran.

6.1 ABSTRACT

SEE ALSO:
Functional Description

A short three to six line functional description.

6.2 AUTHOR

This is the full name of the initial coder of the module. The full name of each maintainer appears in the modification history. Both appear in the module preface.

6.3 CALLING SEQUENCE

SEE ALSO:
Parameters: Formal
Parameters: Input and Output
Procedure

If this routine follows the procedure CALL standard then the calling sequence is:

```
        CALL entry_name (formal parameters)
```

or

```
        value = entry_name (formal parameters)
```

The formal parameters should be documented using the notation in the Functional and Interface Specifications chapter.

If this is a non-standard routine, the method of entry should be given as JSB, INTERRUPT, or EXCEPTION. Any parameters passed in registers or on the stack should be given in the input parameters section. Any parameters left on the stack or in registers should be given in the output parameters section.

6.4 COMMENT

SEE ALSO:
  Comment: Block
  Comment: Documenting
  Comment: Group
  Comment: Maintenance
  Statement: Block

A comment is any text embedded between a <comment delimiter> on the left and the end of the source line on the right.

There is a grey area between the use of too many and the use of too few comments. It is easy to say that there are never enough comments but often there are so many comments that the program text is obscured. In general, comment logically difficult sections of code, structure accesses where it is not clear what is being accessed, and routine invocations, among others. A good rule of thumb is to include a block comment for each block statement.

Above all, strive to comment your program so that anyone can pick it up, read the comments alone and derive a good understanding for what the program does.

In a sense, there are two programs being written; one consisting of code and one consisting of comments. The comment program is written to describe the intent and algorithm of the code. That is, comments are not simply rewordings of the code but are explanations of the overall (gross, if you will) logical meaning of the code.

6.5 COMMENT: BLOCK

SEE ALSO:
  Comment: Group
  Statement: Block

The block comment precedes a block statement, providing reference documentation for the immediately following sequence of statements. A block comment serves to introduce and describe the functionality of a logical grouping of code. It allows the reader to understand the meaning and effect of the code that follows without having to read the code itself. The following rules apply to block comments:

- o The block comment consists of a number of page wide comment lines: The <comment delimiter> is entered, left aligned, in the line's first character position.

- o The first line of the block comment is a begin sentinel, o. the form "!+" or "!++". The single form should be used for internal documentation such as might appear in a program logic manual. The double form should be used for all functional documentation. If the routine is to be part of a general library, the functional documentation should be in a form suitable for publication, see Functional Description.

- o The last line of the block comment is a matching end sentinel, of the form "!- or "!--".

- o The body of the block comment consists of documentary text, separated from the <comment delimiter> by a tab.

- o The block comment is immediately followed by a blank line; immediately following the blank line appears the commented block statement.

Example:

```
<skip>
!+
!       This is a block comment.
!-
<skip>
```

6.6 COMMENT: DOCUMENTING

SEE ALSO:
  Comment: Block
  Module: Preface
  Routine: Preface

The documenting comment is a special format block comment that appears in the module preface and in the routine preface. It serves to describe the functionality of the module and/or routine, as that functionality is to be known from the external point of view: what function is performed, what the input and output parameters are, what values are expected, what completion codes returned, and any other relevant functional information.

- The documenting comment consists of a number of page wide comment lines: the <comment delimiter> is entered in the line's first character position.

- The first line of the documenting comment is a begin sentinel, of the form "!++".

- The last line of the documenting comment is an end sentinel, of the form "!--".

- The documenting comment is structured by means of out-dented keywords that are separated from the <comment delimiter> by a single space. These keywords are part of the standard documenting comment's structure and all of them must be included, in the proper seauence.

- If a specified keyword is not applicable, follow it with the word NONE rather than deleting it. This helps the reader by being explicit about the specification.

- For the body of the documenting comment, see Module Preface, or Routine Preface, or the Program Structure Overview chapter.

Example:

```
!++
!       This is an example of a documenting comment.
!
!       It may be either a module preface, or a routine
!       preface: in each case it has a predetermined format,
!       consisting of a sequence of keywords followed by
!       documentation information.
!--
```

6.7 COMMENT: GROUP

SEE ALSO:
  Comment: Block

Whenever the attention of the reader should be called to a particular sequence of code, a group comment should be used. This might be in any of the following:

1. When several paths join, note the conditions which cause flow to reach this point.

```
;
; All exceptions converge at this point with:
;
;       ...<register and stack status>
;
```

2. At the top of a loop.

```
;
; Loop looking for a handler to call.
;
```

3. When some data base has been built, such as a complex sequence on the stack.

```
;
; At this point the stack has the following format:
;
;       00(SP) = saved R2
;       04(SP) = number of ...
;        ...
;
```

- o The group comment consists of a number of page wide comment lines: the <comment delimiter> is enterred, left aligned, in the line's first character position.

- o The first and last lines of the group comment are just a <comment delimiter> and are set off from surrounding code by a blank line before and after the group. Both the blank comment lines and the blank lines are mandatory and help distinguish the comments and code visually.

- o The body of the group comment consists of descriptive text, separated from the <comment delimiter> by a space.

- o Tabular information is separated from the <comment delimiter> by a tab.

6.8 COMMENT: LINE

A line comment is normally used to explain the meaning of the statement being commented.

A comment is any text following a <comment delimiter>, up to the end of the line. Each and every line of assembly code should be commented.

- o The comment is placed on the right hand side of a non-comment line of text.

- o All assembly language comments are aligned with the <comment delimiter> in column 41 of the text (5 tabs from left margin).

- o The text of the comment is adjacent to the <comment delimiter>.

- o If the statement overflows into the comment field, then its comment is preceded by a space, whereas normally it would be preceded by as many tabs as necessary to position the comment starting with column 41.

- o If the comment is too long to be contained on a single line, or if the statement was too long to be commented on the same line, then the comment may be placed (or continued) on the following line, placing the <comment delimiter> in the same column as the first line and including a space after it.

- o For commenting a multiple-line fragmented statement see statement.

The comment's text should convey the meaning of the associated program text (e.g., instruction MOVAL A,B should be commented "Initialize pointer to first buffer in free area" or such, not "Move the address of A into B".) As a rule of thumb, symbols should not appear in a comment, rather say what the object is or means. If a line of code is totally self evident to the most casual reader then it need not be given redundant commenting text, however it must have a <comment delimiter> (see example). If a comment applies to several successive lines of code, indicate commonality by tagging follow-on lines with comments of the form "!<space> . . .".

As a matter of taste, some coders place a single space after the <comment delimiter>. All modifications to a module should follow the style of the original author. The original source should not be changed to the modifier's style because then a differences listing would be useless.

Example:

```
        STATEMENT                 ;Compute multiple-line function
        STATEMENT                 ; . . .
        STATEMENT                 ; . . .
        OBVIOUS STATEMENT         ;
        STATEMENT                 ;Here we do something new
                                  ; and extend the comment to the
                                  ; next two lines.
        OBVIOUS STATEMENT         ;
        A SOMEWHAT LONG STATEMENT ;And its comment
        A SOMEWHAT LONGER STATEMENT ;And its long comment
                                    ; which continues on
                                    ; additional line(s).
        A VERY VERY VERY VERY VERY VERY LONG STATEMENT
                                  ;And its comment on next line
        A FRAGMENTED            - ;The statement's comment
                STATEMENT         ; . . .
```

6.9 COMMENT: MAINTENANCE

SEE ALSO:
Author
History: Modification
Version Number

When an existing module is modified (as distinct from "originally coded"), each logical unit of modification is assigned a maintenance number in the detailed current history section of the module preface. Use a new number for each logical unit of modification that is being worked on. The maintenance numbers increase by one, are decimal, and are never reset. It is permissable after a release to bump the number to a round number (such as the next 100s) to make room for SPR fixes to follow the release level. Add a maintenance comment --derived from that number-- to each line of source code that is affected. There are two reasons for having maintenance comments:

1. The modifications may well be distributed all over the module. The maintenance comment enables you to find all the places where a correction of a single functional problem was made. This is especially useful if the correction has to be further corrected by someone other than the original modifier and/or if it has to be understood by the software specialist in the field.

2. All too often it happens that as we correct bug "B", we innocently modify an instruction which was the correction for a previous bug "A". Bug "B" is fixed at the expense of the reappearance of bug "A" (or one of its relatives). If modification of a program leads you to the modification of a line that already has a maintenance comment, then find out (from the detailed current history) who the modifier was, consult that person, and exercise extreme caution in effecting your modification.

In many cases the edit numbers may be assigned consistently across all modules in a facility. In this case, the module defining the facility's version number should have a full maintenance history and the others should include only module specific changes.

The following rules apply to maintenance comments:

- o The maintenance comment consists of a <comment delimiter> followed by a code letter, followed by a maintenance number.

- o The code letter may be

  A - this line was ADDED to the text

  D - this line was DELETED. In this case, effect the "deletion" by commenting the line out. Place a <comment delimiter> in the first character position of the line, marking it as a candidate for future physical deletion.

  M - This line was MODIFIED.

- o The maintenance comment is placed after the line's regular comment at column 80

```
        !Regular comment !<maintenance_comment>
```

- o If the modified line already has an existing maintenance comment, then add the new one in front of the existing one

```
        !Regular comment !<new_mc>!<previous_mc>
```

Example:

The maintenance number is assigned in the detailed current history section of the module preface, as follows:

```
! 02    -  SPR #4711: describe the SPR problem
```

The number is now used in maintenance comments for all lines of text affected by the modification called for by SPR #4711:

```
        MODIFIED STATEMENT              !Statement's comment    !M02
        ADDED STATEMENT                 !statement's comment    !A02
!       DELETED STATEMENT               !Statement's comment    !D02
```

NOTE: If the statement is a multiple-line one, make sure to place maintenance comments (or effect a "commenting out" deletion) on all component lines of the statement.

6.10 COMPLETION CODES

The most reliable means for indicating a software detected exception condition occurring in a called procedure is for the called procedure to return a condition value as a function value and for the caller to check the return value for TRUE or FALSE. TRUE is bit 0 set and FALSE is bit 0 cleared. TRUE means that the requested operation was performed successfully; FALSE means an error condition occurred; in both cases, the rest of the value is a condition value. Thus, most procedures are written as functions, rather than subroutines. If it is necessary to indicate an exceptional situation without returning a value, then generate a call to LIB$SIGNAL, see Signals.

The low order three bits, taken together, represent the severity of the error. Severity code values are:

0	Warning
1	Success
2	Error
3	Reserved
4	Severe Error
5-7	Reserved

Bits <31:16> indicate the facility, see the Naming Conventions chapter. Bits <15:3> distinguish distinct conditions or system messages within the facility. Bits <2:0> can vary for a given condition depending upon environment, condition handling, etc. Status codes are expressed in symbolic names in the format:

```
fac$ mnemonic
```

Return status values can be tested by testing the low-order bit of R0 and branching to an error checking routine if the low bit is not set, in the assembler as follows:

```
BLBC    R0,errlabel
```

The error checking routine may check for specific values. It must always ignore <2:0> when checking for a particular condition because <2:0> can vary depending upon the severity in the current environment. For example in assembly language, the following instruction checks for an illegal event flag number error condition:

```
CMPV    #3,#29,R0,#<SS$_ILLEFC@-3>
```

Successful codes other than SS$ NORMAL are defined. In some cases, a successful return includes information about the previous status of a resource. For example, the return SS$ WASSET from the Set Event Flag ($SETEF) system service indicates that the requested flag was already set when the service was called.

6.11 CONFIGURATION STATEMENT

SEE ALSO:
  INCLUDE Files
  Module: Preface

The configuration statement is part of the environment statement in the module preface, and serves to indicate to the programmer how the module is to be assembled. The module may be part of a large system with a system-wide conditional assembly arrangement. It may also have its own peculiar conditional assembly requirements, either alone or in conjunction with system-wide conventions.

State the name(s) of the include file(s) containing conditional assembly parameters (if any). State the conditional assembly variables affecting this module. If the variables are peculiar to this module, state the values that they may assume, and what these values mean.

Example:

```
! ENVIRONMENT:
!
!       This module may be assembled with various parameters
!       changed. This is done by supplying a special copy
!       of the macro $FAC_CHANGE_DEF with the changed symbols in
!       it as a library file. The symbols which can be changed
!       are the default lines per page (DEF_LINES_PPAGE) which
!       is normally 55, and the maximum line width (MAX_LINE_WIDTH)
!       which is normally 132.
!
```

6.12 ENVIRONMENT STATEMENT

SEE ALSO:
  Configuration Statement

This paragraph gives any special environmental assumptions which a module may make. These include both compilation assumptions such as configuration files and execution time such as hardware or software environments. For compile time environments, see Configuration Statement.

For execution time environment describe any situations which the module may assume. For example, it may assume that the hardware is a single processor, or that this module is always invoked with interrupts disabled. The module might assume that it runs only in user mode, that ASTs are disabled, or that storage allocation is handled by the standard procedure library. In general, document here anything out of the ordinary which the module assumes about its environment.

6.13 EXCEPTIONS

SEE Signals

6.14 FACILITY STATEMENT

This section of the module preface gives the full name of the facility of which this module is a part. See the Naming Conventions chapter for a list of the facilities.

6.15 FUNCTIONAL DESCRIPTION

The functional description section of the module and routine prefaces should describe the purpose of the module or routine and should document its interfaces precisely and completely

The functional description should also include the basis for any critical algorithms used. This should include literature references when available. For example, specify why a particular numerical algorithm is used in the math library or why a particular way of sorting was chosen.

The functional description appears in one of three places:

- o As a self-contained short description on the first page of the module and routine prefaces.
- o As the second or more page(s) of the module and routin prefaces. In this case an abstract appears on the first page.
- o As a separate functional specification. In this case an abstract appears on the first page of the module and routine prefaces and a reference to the specification is included.

Example:

```
!++
! FUNCTIONAL DESCRIPTION:
!
! EXP(X) is computed using the following approximation technique:
!
!       If X > 88.028 then overflow
!       If X <= -89.416 then EXP(X) = 0.
!       If |X| < 2**-28 then EXP(X) = 1.
!
! Otherwise,
!
!       EXP(X) = 2**Y * 2**Z * 2**W
!
!       where
!               Y = integer(X*log2(E))
!               V = frac(X*log2(E)) * 16
!               Z = integer(V)/16
!               W = frac(V)/16
!
!               2**W = (P + W*Q) / (P - W*Q)
!
!               P and Q are first degree polynomials in W**2. The
!               coefficients of P and Q are drawn from Hart #1121.
!
!       Powers of 2**(1/16) are obtained from a table.  All
!       arithmetic is done in double precision and then rounded
!       to single precision at the end of calculation. The relative
!       error is less than or equal to 10**-16.4.
!
```

6.16 FUNCTION VALUE

SEE ALSO:
  Completion codes
  Functional and Interface Specification chapter

A function value is returned in register R0 if representable in 32 bits and registers R0 and Rl if representable in 64 bits. If the function value cannot be represented in 64 bits, one of the following mechanisms is used to return the function value:

1. If the maximum length of the function value is known, the calling procedure can allocate the required storage and pass a pointer to the function value storage as the first argument.

   This method is adequate for CHARACTER functions in Fortran and VARYING strings in PL/l.

2. The called procedure can allocate storage for the function value and return in R0 a pointer to a descriptor of the function value.

   This method requires a heap (non-stack) storage management mechanism.

Procedures, such as operating system CALLs, return a success/fail value as a longword function value in R0. Success returns have bit 0 of the returned value set (Boolean true); failure returns have bit 0 clear (Boolean false). The remaining 31 bits of the value are used to encode the particular success or failure status.

6.17 HISTORY: MODIFICATION

SEE ALSO:
  Author
  Comment: Maintenance
  Module: Preface
  Version Number

The detailed modification history is a section of the module preface. An entry is logged for each logical functional modification of the module. For example, if the module is a terminal driver, and bug reports state that sometimes interrupt handling is incorrectly masked and also that deleted characters are handled incorrectly, then these will be given TWO separate log entries: one entry for the interrupt problem, one for the delete problem.

Each log entry is assigned a maintenance number. The maintenance numbers begin with "1" and grow by unit increments. The log entry specifies the maintainer's name, and a description of the problem requiring maintenance.

If a problem that was thought fixed is reopened for further fixes, or if a modification changes hands from one programmer to another, a new log entry (having a new maintenance number) is made.

The maintenance numbers are used to affix maintenance comments at all the places that were modified. This way, it becomes possible for anyone to look at a maintained piece of software (especially anyone in the field) and reconstruct what has happened.

Periodically, at the discretion of the appropriate supervisor, old detailed current history log entries may be deleted, together with their corresponding documenting comments (and lines marked for deletion). It is advised that the deletion not be made until the software has proven itself in the field.

6.18 IMPLICIT INPUTS AND OUTPUTS

SEE ALSO:
  Parameters: Formal
  Parameters: Input and Output
  Side Effects

These sections of a routine preface should include all locations in global or own storage which are read or written by the routine. Any locations which are addressed by parameters should not be documented in these sections, see Parameters: Formal, and Parameters: Input and Output.

ADDITIONAL SPECIFICS TO BE SUPPLIED

6.19 LEGAL NOTICES

A standard DEC copyright statement must always appear on the first page of every source file. It is part of the module preface. The legal notices must be part of the original program text, so that they will be plainly stated on any DEC program listing (regardless of whether the listing was produced by a language processor or was directly printed from the source).

- The legal notices may undergo revision. Make sure that you use the proper current version.
- The legal notices are always in upper case to bring emphasis to them.
- When developing a new module, the year stated is the year of the first release, not of the first coding.
- When modifying an existing program that has legal notices,

  (1) Verify the statements' validity, and

  (2) Add the year of modification to the year stated by the existing copyright statement; DO NOT update that existing year: add the current one (if different), separating it from the last date with a comma (",").

The legal notices are of the following form:

```
!
! COPYRIGHT (C) 1977
! DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
!
! THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
! COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
! ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
! MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
! EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
! TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
! REMAIN IN DEC.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
! CORPORATION.
!
! DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
!
```

The license paragraph should be omitted from software which DEC does not license (e.g., distributed through DECUS or not owned by DEC).

6.20 MODULE

The module is a single body of text that is assembled as a unit. The module is normally part of a larger program or facility that is created by linking all of the component modules object code.

There must be some self evident identity justifying the module's existence. That is to say, the module is not just an arbitrary concoction of code, but a self evident unit of code. Typically, the module consists of either:

- o A single function or database, or
- o A collection of related functions (e.g., all conversion routines) each of which would be too small for an independent module.

The word "module" is used in its hardware sense: a "black box" unit that may be attached or detached, plugged in or out. In order to have this desirable property of a "plug-in module", the module's interface has to be as clean as possible Use formal argument carrying calls for all routines in the module, avoid all functional side effects. In the case of non-standard interfaces, try using a "standard" non-standard interface (i.e., an interface that is uniform within the program of which the module is part.)

The module should contain

THE FUNCTIONALITY,
THE WHOLE FUNCTIONALITY
AND NOTHING BUT THE FUNCTIONALITY!

Then, if it is known that a certain functionality is wholly and exclusively localized to a given module, it becomes possible to replace the module by a more efficient one, or selectively link it into the larger program depending on the runtime requirements. The ability to do this is more useful and important than any local efficiency "hackery' that would jeopardize the module's functional identity. When in doubt, place each routine in a separate module. Combine a few routines primarily when doing so allows own storage to be used rather than global storage. Never combine many routines.

6.21 MODULE: DATA SEGMENT

SPECIFICS TO BE SUPPLIED

6.22 MODULE: FILE NAME

Each module exists as a distinct source text file. The name of the file reflects the module's functionality and also the larger facility of which it may be part.

The module is stored in a filename which is the non-facility part of the name, see the Naming Conventions chapter. The file type is the standard one for the source language. There is no special significance to the file generation version (i.e., it need not match the edit number or increase from release to release). The file is stored in a directory which corresponds to the facility.

6.23 MODULE: PREFACE

The module preface provides uniform documentation of the module. It contains certain control items (TITLE and IDENT) which are needed by the linker, as well as the standard DEC copyright statement needed for the protection of DEC's legal ownership rights. Apart from these items, the module preface contains all of the information that might be needed in order to know what the module is and does, what the module's history is, and how the module relates to the larger software product of which it is a part. This documentation should include the design basis for any critical algorithms.

The module preface is described and illustrated in the Program Structure Overview chapter. All module prefaces should rigorously adhere to the standard format, so that they can be processed mechanically. For example, it should be possible to extract information from the module preface in order to compile technical documentation. This can only be achieved if the module preface is of uniform syntactical construction.

6.24 PARAMETERS: FORMAL

SEE ALSO:
  Implicit Inputs and Outputs
  Parameters: Input and Output
  Procedure
  Routine: Preface

The VAX-11 hardware has a built-in advanced call/return mechanism with provision for automatic argument passing. The caller specifies a list of arguments. The called procedure expects parameters which correspond one-to-one to the caller's arguments.

The procedure's parameters will be bound with the arguments of each caller, at the moment of call. They are known as "formal parameters" because they have no identity (i.e., specific memory address) on their own, but assume the identity of whatever arguments the present caller chooses to supply.

The argument list pointer AP always points at the base of the caller-supplied argument list.

6.25 PARAMETERS: INPUT AND OUTPUT

SEE ALSO:
Calling Sequence
Implicit Inputs and Outputs
Parameters: Formal

These sections of a routine preface should include any parameters passed on the stack or in registers. Any parameters whose locations are addressed directly in own or global storage should be documented as implicit inputs and outputs. Any parameters which are passed via the CALL AP-list mechanism should be documented as formal parameters in the calling sequence.

ADDITIONAL SPECIFICS TO BE SUPPLIED

6.26 PROGRAM

SEE ALSO:
Module
Procedure

An executable program consists of one or more object modules which have been combined and formatted in such a way to be interpretable by an operating system and its hardware.

The following general rules govern the division of program information into modules:

- There is exactly one module within the program, termed the main module, where execution of the program begins.
- If need be, any storage that is referenced by more than one module (i.e., global storage) is declared in one or more modules whose sole purpose is to declare/allocate global storage.
- Separate program operations are divided into modules that contain all of the routines related to a single capability. Examples are symbol table management, binary output generation, and so on.
- Module size is kept moderate in order to facilitate incremental modification and to keep the system resources needed for compilation within reasonable limits.
- When in doubt, place each routine in a separate module.
- Even the main routine is CALLed by an outer environment. Typically this environment is the command interpretter.

6.27 ROUTINE: PREFACE

The routine preface provides uniform documentation of the routine, for the following purposes:

- External functional appearance: From the external point of view, the routine is a "large scale" instruction, performing a high-level function. Like any other instruction, it has to be invoked in a precisely predetermined way and be supplied with arguments of a predetermined form and nature. The routine preface provides exact specifications of the anticipated arguments.

- Runtime behavior: The routine's behavior is dependent on both its input parameter value(s) and possible environmental conditions. For example, the routine OPEN_FILE is dependent on being given a valid file name parameter, as well as on the existence and/or protection of the specified file. It may fail for either reason. The routine's preface specifies the behavior of the routine in case of functional failure: specifies the completion codes that may be returned.

- Side effects: The routine's execution may have functional side effects that are not evident from its invocation interface. Such side effects are documented in the routine's preface. This would include changes in storage allocation, process status, file operations, and signals.

- Functional specification: The short functional specification incorporated in the routine preface should be sufficiently logical and lucid to enable the casual reader to get a fairly accurate idea of what the routine does. This specification should NOT describe HOW the algorithm operates; for that one can read the code (an exception being certain esoteric or elusive effects which otherwise would remain unnoticed from reading the code). The functional specification should explain WHAT the routine's execution accomplishes.

The routine preface is described and illustrated in the Program Structure Overview chapter. All routine prefaces should rigorously adhere to the standard format, so that they can be processed to compile technical documentation.

REMEMBER: It is the CALLed routine which specifies how it is to be called! It is the CALLER'S RESPONSIBILITY to invoke the routine in the precise manner in which it expects to be invoked! The routine preface provides all the necessary information needed in order to determine how a routine is to be called.

6.28 SIDE EFFECTS

SEE ALSO:
  Implicit Inputs and Outputs
  Signals

This section of the routine preface describes any functional side effects that are not evident from its invocation interface. This would include changes in storage allocation, process status, file operations, and signals. In general, document here anything out of the ordinary which the routine does to its environment. If its effect is to modify own or global storage locations, document them as implicit outputs rather than as side effects.

ADDITIONAL SPECIFICS TO BE SUPPLIED

6.29 SIGNALS

SEE ALSO:
Completion Codes
Condition Handler
Side Effects
UNWIND

The most reliable means for indicating a software detected exception condition occurring in a called procedure is for the called procedure to return a completion code as a function value and for the caller to check this return value for TRUE or FALSE. If it is necessary to indicate an exceptional situation without returning a value, then generate a CALL to LIB$SIGNAL to signal the exception. See Appendix D of the System Reference Manual for details on signalling. Current practice is to use this for indicating the occurrence of hardware detected exceptions and for issuing system messages.

When a language or user wishes to issue a signal, it calls the standard procedure LIB$SIGNAL. This routine searches the stack for condition handlers. By convention, the top of the stack normally contains a handler which uses the condition value argument to retrieve a system message from the system message file. It then issues the message to the standard output device. The default handler then takes the default action depending on bit <0> of the condition value. If the bit is set (TRUE) then execution is continued following the call to LIB$SIGNAL. If the bit is clear (FALSE) then execution is terminated and the condition_value is available to the command processor to control execution of the command stream.

When a language or user wishes to issue a signal and never continue, it calls the standard procedure LIB$STOP. This routine is identical to LIB$SIGNAL except that execution never continues.

Thus, the rules for handling exceptional cases in a procedure are very simple:

1. Normally return a completion code to the caller as an indicator of failure.

2. If this is not possible or desirable, issue a message by calling either LIB$SIGNAL or LIB$STOP. Call the former if the signalling procedure can meaningfully continue and the latter if the signalling procedure cannot continue.

3. If the normal situation after issuing the message is to continue execution, then the condition value should have the low order bit set. If the normal situation is to terminate after the message, then the low order bit should be clear.

In addition, the routine LIB$SIGNAL preserves all registers including R0 and R1. Thus, it is possible to insert debugging or tracing signals in a routine without alterring its register usage.

6.30 VERSION NUMBER

SEE ALSO:
Comment: Maintenance
History: Modification
IDENT Statement
Module: Preface

The VAX-11 standard version number is used to provide unique identification of all pre-released, released and inhouse software. It is used both at the module and the facility level. When used for modules, the ident represents the last change made to the module. For facilities which are always bound together such as a compiler, the ident of the module containing the start address is also used as the ident of the facility. The facility (start module) ident must be changed whenever the ident of any component module changes even if the component comes from a library.

The version number is a compound string constructed of the concatenation of the following discrete items:

```
<support> <version> . <update> - <edit> <patch>
```

where:

o <support> is a single capital letter (or null) identifying the support level of the program:

B	benchmark version
D	demonstration version
S	special customer version
T	field test version
V	released or frozen version
X	unsupported experimental version

Typically this letter is omitted from the module ident since it more reflects the program as a whole than any of its modules.

- <version> is a decimal leading zero-suppressed number, starting with "0" and progressing by positive unit increments. Numbers are never skipped. "0" is used prior to the first release. "1" designates the first release, etc. The version identifies the major release, or generation, or base level of a program. It is incremented at the discretion of the responsible supervisor whenever the software has undergone a significant or major change. The module version is incremented upon the first edit after a release so that it reflects the next release.

- <update> if present is a period followed by a single decimal digit indicating a minor release containing internal changes but no significant external changes. Digits are never skipped. Null designates the major release. "1" designates the first update, etc. <update> is cleared when <version> is changed.

- <edit> if present is a minus sign followed by a decimal leading zero-suppressed maintenance number, starting with "1" and progressing by positive unit increments. Numbers may be skipped but may never be lower than that that of a previous edit. The edit identifies any alteration of the source code. It is incremented on every change even if modification history comments are not being kept. Whether <edit> is cleared on release is TO BE SPECIFIED.

- <patch> if present is a single capital letter identifying an alteration to the program's binary object form. The patch character begins with "B" and may be incremented up to "Z", whenever a set of patches is released. This never appears in the source of a module. <patch> is cleared whenever <version> or <update> is changed.

Customers making changes to DEC produced software are advised to follow similar procedures. Customer numbers should be designated by appending a customer version and edit number to the DEC number and putting it inside square brackets.

Examples:

PIP/X3 experiment before third release of PIP

LINK/V5.2-329 released second update to version 5 of LINK;
edit level is 329

LOGIN/V0.3-27 frozen version of LOGIN; part of base level 3
prior to initial release (during initial development);
edit level is 27

RUNOFF/V10.2-527[7-93]
seventh customer version of RUNOFF based on the
second update to the tenth DEC version; DEC
edit level is 527; customer edit level is 93

[End of Chapter 6]

Title: VAX-11 Software Eng. Assembler Formatting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE7R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: none

Author: P. Conklin, P. Marks, M. Spier

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Chapter 7 gives each piece of the assembler formatting and usage conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Split from chapter 6; see chapter 6 for earlier change history.

2. Correct comment column in all examples.

3. Add examples to .IDENT.

4. Add an ident comment to include files.

5. Add common .PSECT statements to description.

6. Limit source line length to 80 columns.

7. Local labels go to 65535.

8. Eliminate single exit point.

9. Change non-CALL to non-standard.

10. Move procedure here from chapter 6.

[End of SE7R3.RNO]

CHAPTER 7

ASSEMBLER FORMATTING AND USAGE

28-Feb-77 -- Rev 3

This chapter contains detailed information on formatting standards, and instruction usage. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

7.1 CALL INSTRUCTIONS

SEE Procedure

7.2 CASE INSTRUCTIONS

SPECIFICS TO BE SUPPLIED

7.3 CONDITIONAL ASSEMBLY

SEE ALSO:
  Configuration Statement

In the example of the configuration statement, the normal definition library for this compilation is assumed to contain a dummy macro named $FAC_CHANGE_DEF which can be superseded by a user supplied one. The default values are defined only if the symbols are not defined by the time the macro has been expanded. This is done in the source file in the equated symbols section:

```
;
; INCLUDE FILES:
;

        $FAC_CHANGE_DEF

;
; EQUATED SYMBOLS:
;

        .IIF NDF DEF_LINES_PPAGE, DEF_LINES_PPAGE=55
        .IIF NDF MAX_LINE_WIDTH, MAX_LINE_WIDTH=132
```

7.4 CONDITION HANDLER

SEE ALSO:
Completion Codes
Signal
UNWIND

For the primary purpose of handling hardware detected exceptions, the VAX-11 system supplies a mechanism for the programmer to specify a handler function to be called when an exception occurs. This mechanism may also be used for software detected exceptions.

Each procedure activation has a condition handler potentially attached to it via a longword in its stack frame. Initially, the longword contains 0, indicating no handler. A handler is established by moving the address of the handler's procedure entry point mask to the establisher's stack frame.

In addition, the operating system provides two exception vectors at each access mode. These vectors are available to declare handlers which take precedence over any handlers declared at the procedure level. These are used, for example, to allow a debugger to monitor all exceptions, whether or not handled. Since these handlers do not obey the procedure nesting rules, they should not be used by procedure library code. Instead, the stack based declaration should be used.

When a condition handler gets control, it is given several arguments. One of these indicates whether the exception occurred in "this" handler's establisher or in a descendant of it. Another argument is the specific condition which occurred. This is in the same form as a completion code and bits <31:3> identify the specific condition.

For further details, see Appendix D of the System Reference Manual. It describes in detail when the handler is called and what its formal parameters are. In addition, the options of the handler are detailed.

7.5 DECLARATION: EQUATED SYMBOLS

SEE ALSO:
  Module: Preface
  Parameters: Formal
  Routine: Preface
  Variables: Stack Local

Define the equated symbols in the proper place as indicated by the module preface and the routine preface sections.

- o Define the equated symbols in alphabetic order if there is no other logical order indicated.

- o If there is some indicated logical ordering, it may be because of either of the following reasons:

    - o Equated symbol A is used in the definition of equated symbol B, hence must have been defined prior to B.

    - o Equated symbols are used to define a based structure, and have to be defined in the order dictated by the structure definition. In this case precede the structure definition with a block comment stating that this is a logical structure definition, and how it is going to be used. See block comment.

- o The equated symbols are defined one per line. The symbol is defined left aligned in the first character position of the line. The definition line has a comment explaining the nature and use of the symbol.

- o A local equated symbol is defined by means of the "=" operator. A global equated symbols is defined by means of the "==" operator.

Example:

```
;
;       Definition of equated symbols
;

CARRET=13                                   ;Carriage return character
FORMFEED=12                                 ;Form feed character
LINEFEED=10                                 ;Line feed character
```

For an example of a structure definition, see structures.

7.6 DECLARATION: VARIABLES

SEE:
  $OWN Macro
  .PSECT Statement
  Structures
  Variables: Stack Local

7.7 DESCRIPTOR

SEE:
  Parameters: Formal
  Functional and Interface Specifications chapter

7.8 EXPRESSIONS

The assembler allows for assembly-time expressions. Typically you will use them when accessing data structures that are relative to some base address. An important reason for using symbols in expressions is so that all references will appear in a cross reference listing.

- o Avoid using absolute numbers in your expressions, especially numbers that are liable to change in the future. Define suitable equated symbols: you will both enhance the readability of your code and facilitate the modification of such numbers without having to change any of your code.

- o When you have recurring expressions, then further equate the expression itself with a mnemonically meaningful symbol.

- o The assembler expression evaluator does not know of operator precedence. Expressions are evaluated in a strict left-to-right order. Make use of angle brackets "< >" (the assembler's notation for algebraic parentheses) to resolve any ambiguity in evaluation precedence.

7.9 $FORMAL MACRO

SEE Parameters: Formal

7.10 .IDENT STATEMENT

SEE ALSO:
Version Number

The .IDENT statement is the second statement of the module. It has, as its parameter, the current version number and edit level of the module separated by a minus ("-"). These numbers correspond to the last entry in the module's modification history.

Example:

```
        .IDENT  /3-47/          edit 47; used in version 3
        .IDENT  /6.2-295/       edit 295; used in version 6.2
```

7.11 INCLUDE FILES

The purpose of INCLUDE files is to centralize in one place declarations and definitions that are common to multiple modules. Data structure declarations, macro declarations, and constant declarations are the principal contents of INCLUDE files.

INCLUDE files are usually in the form of a macro library. In this case, it contains only macro declarations. In order to include structure declarations and constants, the appropriate definitions are included in a structure definition macro. When this macro is called, all the symbols relating to that structure become defined. Refer to the Symbol Naming Conventions chapter for the form of these symbols and the macro name.

The source for INCLUDE files consist of the following:

1. A title comment

   ```
   ; file-name - short description
   ```

2. An ident comment

   ```
   ; .IDENT /6.2-295/
   ```

3. A full set of legal notices.

4. The rest of a module preface to describe the file.

5. The text of the INCLUDE file. The text conforms to the formatting rules for declarations.

6. An end comment

   ```
   ; file-name - LAST LINE
   ```

7.12 INTERLOCKED INSTRUCTIONS

SEE Synchronization: Process

7.13 LABEL

SEE ALSO:
  Label: Local
  Procedure: Entry
  Relative Addressing
  Symbol

A label is a symbol which names a statement. The label is delimited by a colon.

o A label should be meaningful in that it should convey some information about the purpose of the block it precedes.

o Left align all labels in column one of the source text.

o A label should be placed on a line of its own (i.e., not on same line as the labelled item), and be commented unless it is a local label. The comment should explain the logical meaning of the label, and under what circumstances execution reaches the label.

o A statement may sometimes have several (synonymous) labels, in which case they are placed on subsequent lines, and commented individually. NOTE: This practice is generally discouraged. Generally, each item in the program should have at most a SINGLE name. Only in rare cases will a single item justifiably require several names, such as when two distinct functions have been combined.

o The labelled statement is placed on the immediately following line.

Example:

```
   A_LABEL:                                ;Result is Negative
        STATEMENT                       ;Statement's Comment
ANOTHER_LABEL:                          ;Used if GEN SWITCH = OFF
SYNONYMOUS_LABEL:                       ;Used if GEN SWITCH = ON
        STATEMENT                       ;Statement's Comment
```

7.14 LABEL: GLOBAL

SEE ALSO:
  Declaration: Equated Symbols
  Symbol: Global

A global label is declared by means of the double colon "::" operator or in an entry operator.

Example:

```
PRINT::                                         ;Global print routine
        .WORD   ^M<register list>               ;Register save mask
```

or

```
        .ENTRY  PRINT,^M<register list> ;Global print routine
```

7.15 LABEL: LOCAL

SEE ALSO:
LSB: .ENABL/.DSABL

The local label is a special purpose construct "n$:" where "n" is a decimal constant. The value of an explicitly stated "n" may be in the range of integers 1 through 65535 (decimal). Local labels have a limited scope of reference defined by (non-local) label brackets, or by an explicit local symbol block.

- The local label is left aligned in column one of the source text, on the same line as its named statement.

- Local labels serve as necessary but otherwise mnemonically meaningless statement identifiers within a block statement.

- Local labels SHOULD NOT BE USED other then for flow of control identification within a block statement! DO NOT use local labels throughout logically unrelated sequences of statements. If need be, label block statements mnemonically in order to force a change of scope for the following local labels.

- Local labels need be unique only within their given scope; a local label's name may be reused within a new scope.

- Always number your local labels sequentially, from "10$:" upwards by increments of 10 in the order of appearance.

- When inserting a new local label between two existing ones, give it a number within the range of the two existing labels: insert "15$:" between "10$:" and "20$:", "17$:" between "15$:" and "20$:".

- The numbers should be multiples of ten at first release, and should be renumbered on any release which makes extensive changes. They should not be renumbered in the course of maintenance patches or updates.

Example (correct):

```
   LABEL1:                                 ;Begin local label scope
        STATEMENT                       ;
10$:    STATEMENT                       ;
20$:    STATEMENT                       ;

   LABEL2:                                 ;Begin local label scope
10$:    STATEMENT                       ;
```

Example (incorrect):

```
   LABEL1:                                 ;Begin local label scope
50$:    STATEMENT                       ;First label not "10$:"
60$:                                    ;Free standing local label
        STATEMENT                       ;
30$:    STATEMENT                       ;Decreasing label number
120$:   STATEMENT                       ;Increment larger than 10
```

7.16 LIBRARIES

SPECIFICS TO BE SUPPLIED

7.17 LISTING CONTROL

SPECIFICS TO BE SUPPLIED

7.18 $LOCAL MACRO

SEE Variables: Stack Local

7.19 LSB: .ENABL/.DSABL

SPECIFICS TO BE SUPPLIED

7.20 MACROS

SPECIFICS TO BE SUPPLIED

7.21 $OWN MACRO

SEE ALSO:
  Structures

SPECIFICS TO BE SUPPLIED

7.22 PARAMETERS: FORMAL

SEE ALSO:
  Implicit Inputs and Outputs
  Parameters: Input and Output
  Procedure
  Routine: Preface
  Structures
  Variables: Stack Local

The VAX-11 hardware has a built-in call/return mechanism with provision for automatic argument passing. The caller specifies a list of arguments. The called procedure expects parameters which correspond one-to-one to the caller's arguments.

The procedure's parameters will be bound with the arguments of each caller, at the moment of call. They are known as "formal parameters" because they have no identity (i.e., specific memory address) on their own, but assume the identity of whatever arguments the present caller chooses to supply.

The argument list pointer AP always points at the base of the caller-supplied argument list. The first argument list element is accessed as 1*4(AP), and the Nth as N*4(AP). Rather than address those arguments absolutely, define each procedure parameter as a symbolically equated offset relative to AP.

The definition of symbolic formal parameters is made at the end of the routine preface:

```
        $FORMAL         <-              ;
PAR1,-                                  ;PAR1.at.mf is symbolic name
PAR2,-                                  ;PAR2.at.mf is symbolic name
   .                                              .
   .                                              .
PARn>                                   ;PARn.at.mf is symbolic name
```

where the .at.mf specifies the access type, the data type, the passing mechanism, and the passing format. See the Functional and Interface Specifications chapter for more details.

In the body of the procedure, you now refer to the parameters symbolically:

- o Call by reference: refer to the value of the Nth parameter by the form @PARn(AP). Refer to the ADDRESS of the Nth parameter by the form PARn(AP).

- o Call by value: refer to the value of the Nth parameter by the form PARn(AP). You cannot make any meaningful reference to the parameter's address. Warning: the argument list is read only.

- o Call by descriptor: the descriptor is referenced as in call by reference. The structure typically has a more specific referencing algorithm.

Giving the formal parameters symbolic names has the following advantages:

- o The code is readable. The notation @FILNAM(AP) is more meaningful than the notation @12(AP).

- o If it so happens that the procedure's interface has to be changed, and what used to be the Nth argument now is the N+Ith argument, only the parameter definitions have to be revised; the referencing code itself remains unaffected. Moreover, any such modification is made within the routine preface's documenting comment and is thus automatically reflected in the module's documentation.

- o The symbols appear in a cross reference listing.

7.23 PROCEDURE

SEE ALSO:
Parameters: Formal
Routine: Entry: Multiple
Routine: non-standard
Routine: Order

The procedure is a body of code that is CALLed by some other body of code, or recursively by itself, to perform a certain function. The procedure has a certain functional behavior which may be controlled through caller supplied arguments. To the procedure, the caller's arguments are locally known as formal parameters; the procedure does not have to know what the caller's arguments' exact memory address is.

VAX-11 provides one calling mechanism supported by two instructions. The choice of the instruction is strictly up to the caller. The callee always uses AP to reference arguments:

- o The CALLG instruction where the argument list is stored in a caller supplied area, and

- o The CALLS instruction where the argument list has been pushed onto the stack by the caller, immediately prior to the call.

In either case, the argument list itself is read only. By convention, it normally consists of an array of pointers to the actual argument variables. This is NOT mandated by the machine! The argument list may well contain the values of the arguments.

- o According to these conventions, all argument lists by default contain pointers to the argument variables (known as "call by reference").

- o If a procedure is called with argument values ("call by value"), then this fact must be prominently displayed in the procedure preface, in form of a specific notation (see Parameters: Formal).

The procedure may have local variables. Such variables may be either permanently allocated in memory (as a .BLKB, .BLKW or .BLKL allocation) or they may be allocated on the stack (see stack local variables). Stack local variables are allocated upon entry into the procedure, and de-allocated automatically upon return from the procedure. The use of stack locals results in more efficient memory utilization, better working set behavior in the paging environment, and allows the procedure to be called recursively. Even more importantly, stack locals are truly local to the procedure activation and the chance of their values getting clobbered, by some other code that is external to the procedure, is extremely low.

The use of stack locals is recommended. Note that registers are also in the category of stack local variables, assuming that they were specified to be saved in the procedure's entry mask. In general, the only non-stack variables to be used by a procedure are the variables corresponding to some permanent database that the procedure is responsible for maintaining. As a rule, any variable whose value MUST be remembered across procedure call/returns is permanently allocated; all other variables are temporaries and should be stack resident.

7.24 PROCEDURE: ENTRY

SEE ALSO:
  Routine: Entry: Multiple

The procedure entry consists of the procedure name label, and of the procedure entry mask. The first word of a procedure that is called by either CALLG or CALLS is interpreted by the hardware to be a register-save mask. The mask, which is a word (=2 bytes), specifies those registers that are to be saved by the calling mechanism. It also specifies the integer and decimal overflow enables.

You have to specify those registers explicitly. You specify the registers used by your procedure, so that their values will be preserved and restored upon return.

Use the ^M operator to specify the list of registers to be saved:

```
ROUTNAME:                                    ;Name of the procedure
        .WORD    ^M<R2,R3,R4,R10>          ;Save four registers
```

or

```
        .ENTRY  GLOBAL_ROUTNAME,^M<R2,R3,R4,R10> ;Save four registers
```

NOTE: Whenever you modify an existing program, and decide to use a register, carefully verify the fact that the register is specified in the procedure entry's save-mask.

REMEMBER: Being overzealous in specifying "efficient" register save masks may cause bugs which are extremely difficult to find; not necessarily in YOUR procedure, but rather in the procedure that CALLed you. That calling procedure may be from the library, and the bug symptom may be extremely horrible and impossible to trace to YOUR procedure which caused the bug by clobbering the caller's register(s).

If your procedure invokes a non-standard routine your entry mask must specify all registers used by that routine (even if that routine does a PUSHR). This is necessary to allow for the case of a signal or exception being generated and a condition handler UNWINDing the stack. (See Condtion Handler, Signal, and UNWIND.)

7.25 .PSECT STATEMENT

Typically, PSECTs have the following attributes.

Code	PIC	USR	CON	REL	LCL	SHR	EXE	RD	NOWRT	Align(2)
Literals	NOPIC	USR	CON	REL	LCL	SHR	NOEXE	RD	NOWRT	Align(2)
Own	NOPIC	USR	CON	REL	LCL	NOSHR	NOEXE	RD	WRT	Align(2)
Global	NOPIC	USR	CON	REL	LCL	NOSHR	NOEXE	RD	WRT	Align(2)
Common	NOPIC	USR	OVR	REL	GBL	NOSHR	NOEXE	RD	WRT	Align(2)

Since the assembler defaults attributes, the following declarations are sufficient and hence preferred:

Code	.PSECT	name,PIC,SHR,NOWRT,LONG
Literals	.PSECT	name,SHR,NOEXE,NOWRT,LONG
Own/Global	.PSECT	name,NOEXE,LONG
Common	.PSECT	name,OVR,GBL,NOEXE,LONG

Subsequent references to the PSECT should give just the name with no attributes.

7.26 QUEUE INSTRUCTIONS

SEE ALSO:
  Synchronization: Process

SPECIFICS TO BE SUPPLIED

7.27 RELATIVE ADDRESSING

SEE ALSO:
Expressions

The assembler allows the formulation of relative addresses of the form "SYMB+OFFSET". The assembler also allows reference to be made to its current location counter value dot (".").

- Under NO CIRCUMSTANCES is it allowed to make relative address references within the executable code. Code of the form:

  ```
          BR      .+4                         ;This is a NO-NO
  or,
          JMP     LABEL-23                    ;This is a NO-NO
  ```

  is ABSOLUTELY NOT TOLERATED!

- Relative addressing, including dot-relative addressing, is useful --and sometimes necessary-- in the definition of data structures or in the declaration of tables. See expressions, formal parameters and stack local variables for examples.

7.28 ROUTINE: BODY

SEE ALSO:
  Comment: Block
  Procedure
  Statement: Block

The routine's body consists of the sequence of instructions representing the function performed by that routine. The sequence should be decomposed into major groups of instructions, where each group performs a well defined logical operation. Each such group is known as a block statement, and is preceded by its block comment. It should be possible to get a fairly complete knowledge of the routine's logic from simply reading the block comments.

Block statements appear in a logical sequence. The routine's logic must naturally flow in a top-down sequence. All jumps (or branches) must go down the page! The only exception is in the case of loops, where an upwards jump is necessary.

NO SPAGHETTI-BALL CODE IS TO BE TOLERATED!

Note that most loops have their "end" test at the beginning. This is no exception to the above rule in that the loop label is at the top, then the end test including the branch to the exit, then the body followed by the branch back around the loop.

In general, a routine will not have a common exit point because a single RSB or RET instruction performs the return. However, if there is common code in several paths just before return, this should be combined as one exit sequence located at the end of the routine.

7.29 ROUTINE: ENTRY: MULTIPLE

A routine may have several entry points, for either of the following reasons:

- o Two or more outwardly different routines effectively use the same algorithm and have an otherwise identical interface. For example, the routines to convert a binary value into OCTAL, DECIMAL and HEXADECIMAL character representations have a common interface and differ only by the conversion radix.

- o A single function may have two or more variants necessitating different interfaces. For example, both PRINT and PRINT_NL are entries to the routine that prints a line. The first prints the line without a terminating <newline>, the second prints the line and issues a <newline>.

In either case, each entry point is to be documented with a full routine header. Define the entry point, do some setup computation (setting a flag and/or copying the arguments in the case of non-uniform parameters), then transfer to a common label. In the following example, the mandatory routine headers were ommitted for clarity's sake.

Example:

```
;
;       The binary to octal conversion entry
;

        .ENTRY  BIN_TO_OCT,^M<register list> ;Binary to octal
        MOVL    #8,RADIX                     ;Set radix = 8
        BR      common                       ;
<separator>
;
;       The binary to decimal conversion entry
;

        .ENTRY  BIN_TO_DEC,^M<register list> ;Binary to decimal
        MOVL    #10,RADIX                    ;Set radix = 10
        BR      common                       ;
<separator>
;
;       The Binary to hexadecimal conversion entry
;

        .ENTRY  BIN_TO_HEX,^M<register list> ;Binary to hex
        MOVL    #16,RADIX                    ;Set radix = 16
<separator>
COMMON:                                      ;Common conversion code
```

7.30 ROUTINE: NON-STANDARD

SEE ALSO:
  Procedure
  Routine: Preface

The non-standard routine differs from the procedure in the fact that it is invoked with the JSB, BSBB, or BSBW instruction and returns by means of the RSB instruction, whereas the procedure is invoked with either the CALLG or the CALLS instructions and returns by means of the RET instruction.

The non-standard routine has no formal stack frame allocation, nor any hardware supported argument passing mechanism. Arguments are passed in predesignated global localities, most typically in registers or pushed onto the stack.

Code and comment the non-standard routine according to the very same rules laid down for the procedure, as exemplified in the Program Structure Overview chapter. However:

o The non-standard routine's entry point MUST NOT consist of a register save mask. If you have to save registers, use an explicit PUSHR instruction.

o Unlike the RET instruction, the stack does not get cleaned automatically, nor do saved registers get restored automatically. Before performing the RSB instruction, adjust the top-of-stack and perform a POPR instruction (if necessary) to restore the explicitly saved registers (if any).

o In the routine preface, clearly indicate that this is a non-standard routine and not a procedure. Clearly specify where the call arguments are to be found, and in what order (especially important if they are pushed onto the stack). These are documented in the INPUT PARAMETERS section. Similarly document the output registers and stack in the OUTPUT PARAMETERS section.

7.31 ROUTINE: ORDER

The following rules apply to the ordering of routine declarations:

- o All routines appear together as a group and come after all the declarations in a module.
- o Routines are ordered by their use. That is, if routine "A" calls routine "B" then routine "B" appears after "A".
- o Mutually recursive routines are ordered by principal entry first.

7.32 .SBTTL STATEMENT

Whenever you switch from one major logical text element to another, you would normally insert a formfeed to force the new element onto a page of its own (e.g., the module's history, declarative part, and the routine(s)). Begin each such logical element with a .SBTTL statement that will cause that subtitle text then to be reprinted on each successive page of the module element.

If two consecutive logical elements will fit entirely on one page with ample excess space, then the form feed can be replaced by four blank lines. The .SBTTL and comments are always included.

7.33 STATEMENT

SEE ALSO:
  Comment
  Statement: Block

The statement is a single functional step specification of the algorithm. This definition includes functional specifications made to the "assembler machine" as distinct from VAX-11 proper (i.e., assembler directives as distinct from VAX-11 instructions). It also includes higher-level instructions that were defined by means of the MACRO facility.

The statement is of the general form:

```
[LABEL]:                                ;Optional label
        OPCODE  [OPERAND LIST]          ;Opcode and operands
```

Where:

- o [LABEL] is an optional statement label.
- o OPCODE is a VAX-11 Op-Code, or an assembler directive, or a MACRO. It is placed at character position 9 (one tab stop from the left margin).
- o [OPERAND LIST] is an optional list of one or more operands, separated by commas (","). The operand list begins on character position 17 (two tab stops from left margin).

Typically, the statement requires a single line of source text, for example:

```
        MOVL    #10,R5                  ;Initialize loop counter
```

The assembler listing format allows 80 column input lines. VAX-11 instructions, however, may be very lengthy, because:

- o The instruction has a large number of operands, or because
- o The operands themselves are "voluminous".

In addition, because of the object code display constraints, a significant portion of the object listing is dedicated to other than the source text, whose display space is therefore limited. It is therefore very possible that a single statement may not gracefully fit on a single line of text (or even not fit at all).

The statement may be broken into two or more lines of text by means of a statement continuation mark, which is a hyphen ("-"). The mark must be the last non-blank character preceding the comment delimiter. For example:

```
EDIV    BIRTHDAY_CAKE,THREE, -  ;Divide THREE by CAKE
     QUOTIENT,REMAINDER        ;Compute CAKE'th of THREE
```

In general:

- The multiple line statement IS NOT a block statement.

- Use your judgement in best applying the statement continuation feature. It may be put to good use by providing more extensive commenting space on an operand by operand basis, if necessary. Alternatively, there may be good reason to write the statement on a single line (assuming that it fits) and putting the comment on the following line.

- Take pride in producing the most aesthetic looking and consistent source code possible. Having "Raggedy Anne" text and undulating comments is not very pretty. Use the multiple line statement feature to achieve the nicest looking code possible.

- Remember to comment each and every statement. In case that the statement is self evident and needs no comment, remember that a semicolon (";") comment delimiter is still mandatory.

7.34 STATEMENT: BLOCK

A number of statements forming a larger logical unit within the program is known as a block statement. A block statement must not be labelled with a local label (it may include local labels in addition to its own). The block statement need not have a label; however, if it does have local labels then it must be tagged with a label identifying the block.

- o The block statement is separated from its predecessor and successor statements (and/or comments) by a blank line. Its label(s), if it has any, is an integral part of the block statement.

- o The block statement is to be preceded by a block comment.

Example:

```
   <skip>
;+
;       This is the statement's block comment
;-
<skip>
OPTIONAL_LABEL:                            ;Label's comment
        STATEMENT                          ;
10$:    STATEMENT                          ;Optional local labels
        STATEMENT                          ;
<skip>
```

7.35 STRING INSTRUCTIONS

SPECIFICS TO BE SUPPLIED

7.36 STRUCTURES

SEE ALSO:
  $OWN Macro
  Parameters: Formal
  Variables: Stack Local

Structures are allocated under program control. They may appear in the stack, as formal parameters, or at arbitrary places in memory. They are given symbolic offsets from their base and are referenced relative to some base register.

To declare structures, you have to

(1) Define their symbolic offset names, and to

(2) Explicitly allocate space for them.

Example:

```
;
;       Definition of a 3-item based structure
;

ITEM1=0                                 ;ITEM1's offset
ITEM2=4                                 ;ITEM2's offset
ITEM3=8                                 ;ITEM3's offset
ST_LNG=12                               ;Length of this structure
```

- o Assuming memory area VAR to be structured, you will now compute the address of ITEMn by using the expression <VAR+ITEMn>.

- o Assuming the address of the structure to be in base register R1, you will access the first byte of ITEMn by specifying the operand ITEMn(R1).

ADDITIONAL SPECIFICS TO BE SUPPLIED about MDL, SDL, and SYSDEF macros.

7.37 SYMBOL

A symbol is an alphanumeric string of up to 15 characters in length. It consists of letters "a" through "z" and "A" through "Z", digits 0 through 9, and special characters underline ("_"), dot (".") and currency sign ("$").

- o The assembler does not distinguish between upper- and lower-case alphabetic characters constituting a symbol. Thus "symbol", "SYMBOL", "SyMbOl", "sYmBoL" etc. are all interpreted as equivalent. To minimize reader confusion, never use lower case in symbols. Lower case should be used only in comments and in text strings.

- o The underline character "_" is used to separate the parts of a compound (or qualified) name. Freely use the underline when constructing names to improve readability and comprehension.

- o The ability of a programmer to infer various attributes of a symbol simply by virtue of its name is a very desireable characteristic.

- o The currency sign "$" has been given a special significance within the global VAX-11 software architecture.

Refer to the Naming Conventions chapter for the exact symbol construction rules.

7.38 SYMBOL: EXTERNAL

External symbols will be declared automatically by the assembler. A declaration is needed only if the reference is to be weak (see .WEAK Declaration).

7.39 SYMBOL: GLOBAL

SEE ALSO:
  .VALIDATE Declaration
  .WEAK Declaration

A global symbol is defined by means of the double colon "::" for label symbols, and by means of the double equate "==" for equated symbols.

Example:

```
   SWITCH::
        .BLKW   1                           ;Global variable SWITCH
TRUE==1                                     ;Global value TRUE
```

7.40 SYNCHRONIZATION: PROCESS

SEE ALSO:
  QUEUE Instructions

SPECIFICS TO BE SUPPLIED

7.41 .TITLE STATEMENT

SEE ALSO:
  Module: Preface

The .TITLE statement is the very first statement of the module. Its operand is the module name. Any text following the module name is used in the header of the object code listing. The text following the module name should be a terse functional description of the module.

Example:

```
           .TITLE  FILE_MGR - The STARLET file manager subsystem
```

7.42 UNWIND

SEE ALSO:
  Condition Handler
  Signal

If a condition handler gets control, it has several options over the flow of control. It can resignal the condition for another handler to take control, or it can signal a distinct condition for the same purpose. Alternatively, it can continue from the signal. The final option is to terminate the procedures in progress, unwind the stack, and branch to a specific recovery address. This would be done when the current operation is to be aborted, but the program is not to be terminated.

When an unwind is requested, each stack frame is examined in order to restore all the saved registers and Program Status Word (PSW). Before each stack frame is removed, it is examined to see if a condition handler has been established. If so, the handler is called first. This allows a procedure to gain control if it is aborted or if any routine below it aborts. This might be used, for example, to release any resources such as dynamic storage which the routine might have acquired.

7.43 .VALIDATE DECLARATION

SEE ALSO:
  Symbol: External
  Symbol: Global
  .WEAK Declaration

This is used in addition to a global declaration for any symbol which is made global only to validate consistency across several modules. For example, if two modules assume that the length of a particular structure is 47, then both might declare

```
        .VALIDATE STR_LEN
STR_LEN==47
```

This would cause the LINKER to validate that both declarations are the same. The .VALIDATE declaration should not be made if any routine references STR_LEN as an external. It is used only to mark global definitions whose purpose is totally redundant.

7.44 VARIABLES: STACK LOCAL

SEE ALSO:
  Expressions
  Parameters: Formal
  Structures

Stack local variables are allocated at the base of the procedure's stack frame, and given symbolic names that are offsets relative to the procedure's stack frame pointer FP.

Variables may be allocated starting with the longword following FP (the word that would be used by a PUSHL instruction).

To declare stack local variables, you have to:

(1) Define their symbolic offset names, and

(2) explicitly allocate space for them on the stack.

Symbolic definition is performed using the $LOCAL macro, as in:

```
;
;       Definition of stack local variables
;

        $LOCAL  <-
<I,8>,-                                 ;Quad variable I
J,-                                     ;Long variable J
<K,2>,-                                 ;Word variable K
<B,1>>                                  ;Byte variable B
```

The actual allocation is performed using a SUBL2 instruction, as in:

```
;
;       The routine entry point
;

ROUTNAME:                               ;The routine's name
        .WORD   ^M<register list>       ;Save mask
        SUBL2   #$$LOCAL_SIZE,SP        ;Advance SP past allocation
```

Whenever you want to reference one of the stack local variables, do so by using its symbolic name VAR based on the contents of FP (e.g., "VAR(FP)"). Such as:

```
        MOVB    R7,B(FP)                ;Store byte in local B
        ADDL3   I(FP),4+I(FP),J(FP)     ;Add both halves of I into J
```

Compare the allocation of these local variables to the structure definition shown in the structures section. Notice the difference that is due to the stack's backwards growth.

7.45 .WEAK DECLARATION

SEE ALSO:
.VALIDATE Declaration

The .WEAK declaration can be made on either external or global definitions. In both cases its meaning is that the symbol should be matched by the LINKER if defined, but that this reference or definition should not force the loading of a library module.

When used on a global declaration, then the definition of the symbol in this module is not sufficient to cause this module to be loaded from a library. Thus, it should be used for any subordinate symbols defined in a library module.

When used on an external, then the reference to this symbol will not cause it to be defined by loading a library module. If some module which is loaded defines the symbol, then it will be defined for this reference. If nothing defines the symbol, it is automatically satisfied as defined as 0 without any error messages. Thus, it can be used to establish a pointer to an optional module or data base. If the module is loaded, the pointer is defined. Otherwise the pointer has value 0.

[End of Chapter 7]

Title: VAX-11 Software Eng. BASIC Formatting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE8R3.RNO

PDM #: not used

Date: 23-Feb-77

Superseded Specs: none

Author:

Typist: P. Conklin

Reviewer(s):

Abstract: Chapter 8 gives each piece of the BASIC formatting and usage conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date

CHAPTER 8

BASIC FORMATTING AND USAGE

23-Feb-77 -- Rev 3

This chapter contains detailed information on formatting standards, and instruction usage. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

THE CONTENTS ARE TBS

[End of Chapter 8]

Rev 2 to Rev 3:

1. Create null chapter.

[End of SE8R3.RNO]

Title: VAX-11 Software Engineering BLISS Formating and usage
Specification Status: draft

Archetectural Status: under eco control

File: SE9R3.RNO

PDM: not used

Date: 21-Feb-77

Superceded specs: none

Author: P. Marks, M. Spier

Typist: G. Hesley, R. Murray

Reviewer(s): D. Cutler P. Conklin R. Gourd I. Nassi S. Poulsen

Abstract: This chapter is a collection of procedures and examples of specific BLISS related formats and language usages. It is organized by keywords, in alphabetical order.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	P.Marks, M.Spier	2-Aug-77
Rev 2	Review	P.Marks,I.Nassi	1-Jan-77
Rev 3	SEM integration	R.Murray	31-Feb-77

Rev 2 to Rev 3:

1. split from chapter 6 to exclude those features common to both Bliss and Assembler.

[end of se9r3.rno]

CHAPTER 9

BLISS FORMATING AND USAGE

21-Feb-77 -- Rev 3

The following is an explanation of some of the terms used throughout this section.

Logical tab
equivalent to four (physical) spaces. Used for indenting BLISS source text. Two successive logical tabs should be typed as one physical tab.

Physical tab
the ASCII TAB character (octal 11). All standard DEC software interprets the tab as equivalent to moving the carriage or cursor to the next column number which is one more than a multiple of eight.

Tab
used throughout this manual to mean logical tab.

Indentation level
the number of logical tabs a line of text is offset to the right of the left page margin.

Indented
offset one logical tab to the right of the text on the preceding line.

Line
The contents of one record.

9.1 DECLARATION

See:

Declaration: Format
Declaration: FORWARD ROUTINE
Declaration: MACRO
Declaration: Order

9.2 DECLARATION: FORMAT

Declarations are written according to the following format:

```
        declaration-keyword(s)
                declaration-item,                 ! Comment
                declaration-item,                 ! Comment
                    ...
                    ...
                declaration-item;                 ! Comment
```

The following rules apply to declaration formatting:

- o Each declaration-keyword appear(s) alone on a line and starts at the left margin of the block in which the declaration is being made.

- o The declaration-item(s) being declared appear indented one logical tab with respect to the declaration-keyword and on a separate line(s).

- o Declaration-items are in an order meaningful to the program organization, or in alphabetical order.

- o Each declaration-item has a line comment on the same line describing, in most cases, the meaning and/or usage of the declaration-item being declared.

9.3 DECLARATION: FORWARD ROUTINE

SEE ALSO:
Declaration: Format

The following rules apply to FORWARD ROUTINE declarations:

- o forward ROUTINE declarations for a module are grouped together and appear at the beginning of the module.
- o The FORWARD ROUTINE declaration names all the routines to be declared in the module in order of occurrence.
- o Each routine name is on a separate line with a line comment briefly explaining its function.
- o The FORWARD ROUTINE declaration serves as a table of contents for the module.

9.4 DECLARATION: MACRO

SEE ALSO:
Declaration: Format
Expression

The following rules apply to MACRO declarations:

- o MACRO declarations follow the general formatting rules outlined under DECLARATION: format.
- o If the body of the MACRO is composed of declarations and/or expressions, then the body conforms to all the formatting rules for declarations and/or expressions.
- o If the macro has a formal-list, then the commenting rules for ROUTINES should be applied, in so far as describing each of the formal parameters and commenting on the function of this macro.

9.5 DECLARATION: ORDER

SEE ALSO:
Declaration: FORMAT
Declaration: FORWARD
Declaration: MACRO
Routine

We group the BLISS declarations as follows:

1. FORWARD declarations

2. REQUIRE declarations

3. All other declarations

4. ROUTINE declarations

The first, second and fourth groups are discussed in their own sections. The third group lumps all other declarations (e.g., STRUCTURES, LITERALS, MACROS, etc.), which have module-wide or routine-wide scope, into one major group.

The ordering of the different declarations within this third group is important and is based on the following rules:

- Group logically related declarations together. For example, a specific structure may be used in conjunction with certain macros. These declarations would then appear together as a group.

- As much as possible, these logical groups will appear in the order of their use within the module or routine.

- Separate the logical groups from each other by the use of appropriate separators.

- Within a logical group of declarations group specific declarations together by type. For example, all MACROS will be defined via one or more MACRO declarations.

A word of caution: Owing to the nature of the BLISS language, it is necessary to declare all variables, structures, routines, etc. before they are used. Care should be taken so as not to use something before it is declared. In any event, the compiler will complain.

9.6 EXPRESSION

SEE:
Expression: Assignment
Expression: CASE
Expression: Block
Expression: Format
Expression: IF/THEN/ELSE
Expression: INCR/DECR
Expression: SELECT
Expression: WHILE/UNTIL/DO

9.7 EXPRESSION: ASSIGNMENT

Assignment expressions are usually of the form:

```
                name = expression
```

The following rules apply to assignment expressions:

- o If the entire assignment expression will not fit on one line because of its length then place the variable and the equal sign on one line and continue the expression indented one logical tab on the next line.

Examples (correct):

```
        name = a-short-expression;                 ! comment

                (Note the space before and after the
                 = sign.)

        name = a-short-expression;                 ! a long long
                                                   ! comment

        name =                                     ! a comment for
            a-long-long-long-long-expression;      ! this expression
```

9.8 EXPRESSION: CASE

CASE expressions are set up according to the following skeletal example:

```
                CASE  index
                    FROM low-case TO high-case OF
                    SET

                    case-label-action:

                    case-label-action;

                    ...
                    ...

                    case-label-action;

                    TES
```

where case-label-action is:

```
                    [case-label]:
                        ! Explanatory comments
                        ! for this case.

                        case-action;
```

or

```
                    [case-label]: case-action;          ! comment
```

The following rules apply to CASE expressions:

- o The body of the CASE expression is indented one logical tab with respect to the keyword CASE.
- o Each case-label-action is separated from another case-label-action by at least one blank line.
- o The choice of format for the case-label-action is dependent on the size (number of expressions) of the case-action. A large case-action will use the first format; a small case-action the second format.
- o Each of the case-actions follows the rules for expression formatting.
- o It is desirable that the case-label be a descriptive and meaningful name that has been bound to its value. A case-label then becomes a label or signal to the reader indicating what value caused this case-action to be used.

9.9 EXPRESSION: BLOCK

A block expression provides a means of grouping declarations and/or expressions into a single structural entity.

The following rules apply for BLOCK expressions:

- The block expression is separated from its predecessor and successor expressions (and/or comments) by a blank line.
- The block expression is to be preceded by a block comment.
- Constituent declarations and expressions of a block are indented to the same level as the BEGIN-END delimiters.
- In a block expression, the last expression in the block is followed by a ";" unless the value of the block expression is actually used in an enclosing expression.

9.10 EXPRESSION: FORMAT

Specific formatting rules apply for each kind of executable expression. In general, the following rules apply:

- Expressions generally appear on separate lines.
- Expressions are left justified to the current indentation level.
- Expressions which fit on one line may appear on one line.
- Expression subparts, when indented, are indented one logical tab to the right of the start of the expression. Specific indentation rules are given in the appropriate sections.
- Compound-expressions consisting of more than one line are bounded by BEGIN-END delimiters rather than by parentheses.
- In general, for arithmetic expressions:
  - Place one space around the binary "+" and "-".
  - Place one space before the unary "+" and "-".
  - Place no spaces around the "*" and "/" operators.
  - In lists, place one space before the "(" and one space after each "," and the ")".

9.11 EXPRESSION: IF/THEN/ELSE

IF expressions are written in either of two formats:

```
IF test THEN consequence ELSE alternative
```

or

```
IF test
THEN
    consequence
ELSE
    alternative;
```

- In the first case, the entire IF expression may be placed on one line only if the IF expression fits on one line.
- Otherwise, the second format is used. The consequence and alternative expressions are indented one logical tab with respect to the keyword IF.

If the test is a compound test then the IF expression is written in one of the following manners:

```
IF test AND test AND test
THEN
    consequence
ELSE
    alternative
```

or

```
IF test AND
   test AND
   test
THEN
    consequence
ELSE
    alternative
```

- The first format is used when the compound test can fit on one line. Otherwise, the second format is used.

9.12 EXPRESSION: INCR/DECR

INCR/DECR expressions are written according to one of the following formats:

```
        INCR loop-index FROM first TO last BY step DO
            loop-body;
```

or

```
        INCR loop-index
            FROM first TO last BY step DO
            loop-body;
```

The following rules apply to INCR/DECR expressions:

- o Use the first format when the FROM-TO-BY expression will fit on one line. Otherwise, use the second format.
- o The loop-body is indented one logical tab with respect to the keyword INCR/DECR.

9.13 EXPRESSION: SELECT

SELECT expressions are set up according to the following skeletal example:

```
SELECT select-index OF
    SET

    select-label-action;

    select-label-action;

            ...
            ...

    select-label-action;

    TES
```

where select-label-action is:

```
[select-label]:
    ! Explanatory  comments
    ! for this select-label.

    select-action
```

or

```
[select-label]: select-action;       ! comment
```

The following rules apply to SELECT expressions:

- The body of the SELECT expression is indented one logical tab with respect to the keyword SELECT.
- Each of the select-label-action expressions is separated by at least one blank line.
- The choice of format for the select-label-actions is dependent on the size (number of expressions) of the select-action. A large select-action will use the first format; a small select-action the second format.
- It is desirable that the select-label be a descriptive and meaningful name that has been bound to its value. A select-label then becomes a label or signal to the reader indicating what condition or value caused this select-action to be SELECTed.

9.14 EXPRESSION: WHILE/UNTIL/DO

WHILE/UNTIL/DO expressions are written in the following manner:

```
WHILE test DO
    loop-body;
```

or

```
DO
    loop-body
WHILE test;
```

The following rules apply to WHILE/UNTIL/DO expressions:

- The keyword WHILE or UNTIL is aligned with the current indentation level.
- The loop-body is indented one logical tab with respect to the keyword WHILE or UNTIL and follows the rules for expression formatting.

9.15 IDENT MODULE SWITCH

The IDENT switch has, as its parameter, the current version number of the module. This version number corresponds to the last entry in the module's ABBREVIATED HISTORY.

9.16 LABELS

A label is a name, hence it must conform to the rules for constructing names. It is delimited by a colon ":".

The following rules apply to labels:

- o Labels, when used, appear alone on a line. The block to which they refer follows on the next line indented one logical tab with respect to the label.
- o A label is meaningful in the sense that it conveys some information about the block it is labelling.

9.17 MODULE: SWITCHES

Module switches appear in the module declaration and allow the programmer to provide information about the module and to control some aspects of the compiler's treatment of the module. Of special importance is the IDENT switch (see IDENT Module Switch) and the MAIN switch which specifies which routine is to be used to begin program execution.

- o Each module switch will appear on a line by itself. The IDENT switch is first, the MAIN switch is second; any other switches follow.

Example (correct):

```
        MODULE  EXAMPLE (
                            IDENT = '03',
                            MAIN  = BEGINHERE,
                            RESERVE = (R0, R1)
                           ) =
```

9.18 NAME

A name consists of one to fifteen characters from the sets:

1. A B C D E ... X Y Z

2. a b c d e ... x y z

3. 0 1 2 3 4 5 6 7 8 9

4. underline "_"

5. dollar "$"

No distinction is made between upper and lowercase letters except in string literals. Thus, Date_Of_Birth is equivalent to date_of_birth.

The following rules apply to names:

- Freely use the underline "_" when constructing names to improve readability and comprehension. For example: WRITEARECORD becomes WRITE_A_RECORD.

- The ability of a programmer to infer various attributes of a symbol simply by virtue of its name is a very desirable characteristic.

- Predefined and syntactically meaningful names are to be used only for their intended purpose.

9.19 REQUIRE FILES

The purpose of REQUIRE files is to centralize in one place declarations and definitions that are common to multiple modules. Data STRUCTURE declarations, MACRO declarations, and LITERAL declarations are the principal contents of REQUIRE files.

REQUIRE files consist of the following:

1. ! file-name - description

2. A copyright statement and disclaimer.

3. A MODULE PREFACE.

4. The text of the REQUIRE file. The text conforms to the formatting rules for declarations.

5. ! file-name - LAST LINE

9.20 ROUTINE

SEE:

Declaration: Order
Routine: Format
Routine: Name
Routine: Order
Routine: Preface

9.21 ROUTINE: FORMAT

The following rules apply for ROUTINE formatting:

- The routine declaration is to start at the left margin.
- The routine body is to be indented one logical tab to the right of the routine declaration.
- All other indentation follows the rules for declaration and/or expression formats.

9.22 ROUTINE: NAME

Global routine names should follow the naming conventions stated earlier. Local routine names may be chosen at as desired.

9.23 ROUTINE: ORDER

The following rules apply to the ordering of routine declarations:

- All routine declarations appear together as a group and constitute the last set of declarations in a module.
- Routines are ordered by their use. That is, if routine "A" calls routine "B" then routine "B" is declared after "A".

- o The ordering of routines is reflected in the FORWARD declaration group appearing at the beginning of the module.

- o Mutually recursive routines are ordered by principle entry first.

9.24 STRUCTURE: DECLARATION

SEE:
    STRUCTURE: Block

The format for the structure declaration is as follows:

```
        STRUCTURE
            structure-name [access formal list;allocation formal list]=
                [structure size]
                structure body;
```

The following rules apply to the structure declaration:

- o The structure declaration format generally conforms to that of macros

- o The structure-name is indented one logical tab.

- o The structure size and structure body are indented another logical tab.

- o The structure body contains one expression. The format rules regarding expressions are in force starting with the indicated indentation level.

In the instance where the expression part of the structure body is simple, it may be contained on one line as seen below:

```
        STRUCTURE
            BLOCK[O,P,S,E;N,UNIT = %UPVAL] =
                [N*UNIT]
                (BLOCK+O*UNIT)<P,S,E>;
```

or, may be of such complexity as to require the use of most rules for formatting expressions.

```
        STRUCTURE
            VECTOR1CH[I;N,UNIT = %UPVAL] =
                [N*UNIT]
                BEGIN
                    LOCAL T;
                    T=.I;
                    IF .T LSS 1 OR .T GTR N
                    THEN
                        BEGIN
                        ERROR(.T);
                        T=1;
                        END;
                    VECTOR1CH + (.T - 1) * UNIT
                END;
```

9.25 STRUCTURE: BLOCK

SEE:
    STRUCTURE: Declaration

The structure called BLOCK is a predeclared structure which may be used without an explicit declaration. If declared it would look as follows:

```
        STRUCTURE
            BLOCK[O,P,S,E;N,UNIT = %UPVAL] =
                [N*UNIT]
                (BLOCK + O * UNIT)<P,S,E>;
```

Consider the following example.

```
        OWN
                X:BLOCK[2];
                .
                .
                .
        A = .X[0,0,16,0];
        B = .X[0,16,16,0]
        C = .X[1,0,32,0]
```

X is defined as a two word BLOCK whose first word has two fields, each 16 bits long and whose second word is a field 32 bits long. The above assignment statements use the BLOCK definition to access each field.

NOTE

For a further explination of the structure declaration and built-in structures, see the chapter on Data Structures in the BLISS Language Guide.

The BLISS programmer is strongly urged to hide the 4-tuple used to access a BLOCK by using a "field macro" as follows:

```
MACRO
    FIELD_ONE = 0,0,16,0%,
    FIELD_TWO = 0,16,16,0%,
    FIELD_THREE = 1,0,32,0%;
```

Thus the access to the BLOCK X becomes:

```
A = .X[FIELD_ONE];
B = .X[FIELD_TWO];
C = .X[FIELD_THREE];
```

This achieves a greater degree of readibility and facititates future changes to the structure of X.

[end of chapter 9]

Title: VAX-11 Software Eng. COBOL Formatting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE10R3.RNO

PDM #: not used

Date: 23-Feb-77

Superseded Specs: none

Author:

Typist: P. Conklin

Reviewer(s):

Abstract: Chapter 10 gives each piece of the COBOL formatting and usage conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date

Rev 2 to Rev 3:

1. Create null chapter.

[End of SE10R3.RNO]

CHAPTER 10

COBOL FORMATTING AND USAGE

23-Feb-77 -- Rev 3

This chapter contains detailed information on formatting standards, and instruction usage. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

THE CONTENTS ARE TBS

[End of Chapter 10]

Title: VAX-11 Software Eng. Fortran Formatting -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE11R3.RNO

PDM #: not used

Date: 23-Feb-77

Superseded Specs: none

Author:

Typist: P. Conklin

Reviewer(s):

Abstract: Chapter 11 gives each piece of the Fortran formatting and usage conventions in detail. The items are in alphabetical order. Each item includes references to related topics, gives the background and the rules, and then gives templates and examples.

Revision History:

Rev #	Description	Author	Revised Date

Rev 2 to Rev 3:

1. Create null chapter.

[End of SE11R3.RNO]

CHAPTER 11

FORTRAN FORMATTING AND USAGE

23-Feb-77 -- Rev 3

This chapter contains detailed information on formatting standards, and instruction usage. For ease of reference, it is organized alphabetically by topic. Each topic includes references to related topics. Most entries also include examples or sample templates illustrating the specific topic.

THE CONTENTS ARE TBS

[End of Chapter 11]

Title: VAX-11 Software Engineering Naming Conventions -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE12R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: VAXS notes; based on STARLET Working Design Document

Author: P. Conklin, S. Gault

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, D. Tolman

Abstract: Chapter 12 gives the system wide naming conventions for all public symbols. These rules are to be followed by all DEC software for all symbols which are global or appear in parameter definition files. This chapter also includes the list of all facility prefixes.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	S. Gault	Oct-76
Rev 2	Revised from Review	S. Gault	Jan-77
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Add table of prefixes.

2. Add reasons for the rules.

3. Add BLISS field extract macro names. Add .PSECT names. Add non-CALL entry names. Change $C_ to $K_ for constants.

4. Add transportable data types of A, C, G, H, and U. Note reservations of I and R for specific purposes, use of X and Y for context dependent purposes, use of Z for unspecified or nonstandard forms, use of N and P for decimal strings, and O as a general escape valve.

5. Add all known facility prefixes.

6. Reserve data type J to customers.

7. Note reserved status codes<2:0>. Note that <31:16> indicate facility. Add facility codes to section 7.3.

8. Change non-call routine name pattern to agree with OTS.

9. Change BLISS field reference mnemonics. Reserve E to DEC.

10. Clarify that numeric string is all byte forms.

11. Add argument style column to facility table.

12. Clarify that system macro names are general and don't have the facility name.

13. Clarify that BLISS field names have offset, position, size, and sign.

14. Clarify that assembler V symbols are within containing field.

15. Clarify that masks are not right justified.

16. Add facility to structure def macros.

17. Define sizes of transportable codes for reference. Change H to be good for counters (16 to 18 bits).

18. Add B32, FAB, IO, NAM, NET, PLI, RAB, RM, SWP, TST, XAB prefix. Remove CHF prefix.

19. Ban local synonyms for public symbols.

20. Move completion code description to chapter 6.

21. Clarify that H is integer.

22. Clarify that the N and P count is a digit count.

23. Clarify private symbol usage.

24. Add facility codes for all procedure library facilities.

[End of SE12R3.RNO]

CHAPTER 12

NAMING CONVENTIONS

28-Feb-77 -- Rev 3

The conventions described in this chapter were derived to aid implementors in producing meaningful public names. Public names are all names which are global (known to the linker) or which appear in parameter or macro definition files and libraries in more than one facility.

These public names are all constrained to follow these rules for the following reasons:

- By using names reserved to DEC, we ensure that customer written software will not be invalidated by subsequent releases of DEC products which add new symbols.

- By using definite patterns for different uses, we allow the reader to judge the type of object being referenced. For example, the form of macro names is different from offsets, which is different from status codes.

- By using certain codes within a pattern, we associate the size of an object with its name. This increases the likelihood that the reference will use the correct instructions.

- By using a facility code in symbol definitions, we give the reader an indication of where the symbol is defined. We also allow separate groups of implementors to choose names which will not conflict with one another.

Never define local synonyms for public symbols. The full public symbol should be used in every reference to give maximum clarity to the reader.

12.1 PUBLIC SYMBOL PATTERNS

All DEC public symbols contain a currency sign. Thus, customers and applications developers are strongly advised to use symbols without currency signs to avoid future conflicts.

Public symbols should be constructed to convey as much information as possible about the entity they name. Frequently, private names follow a similar convention; the private convention then is the same as the public one with an underline instead of the currency sign. These are used both within a module and globally between modules of a facility which is never in a library. All names which might ever be bound into a user's program must follow the rules for public names; in the case of undocumented names a double currency sign convention can be used such as in 3 below.

Public names are of the following forms:

1. Service macro names are of the form:

   $macroname

   A trailing _S or _A distinguishes the stack and separate arglist forms. These names appear in the system macro library and represet a call to one of many facilities. The facility name usually does not appear in the macro name.

2. Facility specific public macro names are of the form:

   $facility_macroname

3. System macros which use local symbols or macros always use ones of the form:

   $facility$macroname

   This is the form to be used for symbols generated by a macro and used across calls to it and for internal macros which are not documented.

4. Status codes and condition values are of the form:

   facility$_status

   See completion codes in the Commenting Conventions chapter.

5. Global entry point names are of the form:

   facility$entryname

6. Global entry point names which have non-standard calls are of the form:

   ```
   facility$entryname_Rn
   ```

   where registers R0 to Rn are not preserved. Note that the caller of such an entry point must include at least registers R2 through Rn in its own entry mask.

7. Global variable names are of the form:

   ```
   facility$Gt_variablename
   ```

   The letter G stands for global variable and the t is a letter representing the type of the variable as defined in the next section.

8. Addressable global arrays use the letter A (instead of the letter G) and are of the form:

   ```
   facility$At_arrayname
   ```

   The letter A stands for global array and t is one of the letters representing the type of the array element according to the list in the next section.

9. In the assembler, public structure offset names are of the form:

   ```
   structure$t fieldname
   ```

   The t is a letter representing the data type of the field as defined in the next section. The value of the public symbol is the byte offset to the start of the datum in the structure.

10. In the assembler, public structure bit field offset and single bit names are of the form:

    ```
    structure$V_fieldname
    ```

    The value of the public symbol is the bit offset from the start of the containing field (not from the start of the control block).

11. In the assembler, public structure bit field size names are of the form:

    ```
    structure$S_fieldname
    ```

    The value of the public symbol is the number of bits in the field.

12. For BLISS, the functions of the symbols in the previous three items are combined into a single name used to reference an arbitrary datum. Names are of the form:

    structure$x_fieldname

    where x is t for standard sized data and x is V for arbitrary and bit fields. The macro includes the offset, position, size, and sign extension suitable for use in a REF BLOCK structure. Most typically, this name is definable as

```
MACRO
    structure$V_fieldname =
        structure$t_fieldname,
        structure$V_fieldname,   !assembler meaning
        strucutre$S_fieldname,
        <sign extension> %;
```

13. Public structure mask names are of the form:

    structure$M_fieldname

    The value of the public symbol is a mask with bits set for each bit in the field. This mask is not right justified; rather it has structure$V_fieldname zero bits on the right.

14. Public structure constant value names are of the form:

    structure$K_constantname

15. .PSECT names are of the form:

    facility$mnemonic

16. Module names are of the form:

    facility$mnemonic

    The module is stored in a file with filename "mnemonic" in a directory corresponding to the facility.

17. Public structure definition macro names are of the form:

    $facility_structureDEF

    Invoking this macro defines all the structure$xxx symbols.

Example of usage:

IOC$IODONE Entry point of the routine IODONE in the I/O subsystem.

UCB$B_FORK_PRI Offset in the UCB structure to a byte datum containing the fork priority.

UCB$L_STATUS   Offset in the UCB structure to a longword datum containing status bits.

CRB$M_BUSY   Mask pattern for the busy bit in the CRB structure.

CRB$V_BUSY   Bit offset in the CRB structure of the busy bit.

12.2 OBJECT DATA TYPES

The following are the letters used for the various data types or are reserved for the following purposes:

letter	data type or usage
A	address (*)
B	byte integer
C	single character (*)
D	double precision floating
E	reserved to DEC
F	single precision floating
G	general value (*)
H	integer value for counters (*)
I	reserved for integer extensions
J	reserved to customers for escape to other codes
K	constant
L	longword integer
M	field mask
N	numeric string (all byte forms)
O	reserved to DEC as an escape to other codes
P	packed string
Q	quadword integer
R	reserved for records (structure)
S	field size
T	text (character) string
U	smallest unit of addressable storage (*)
V	field position (assembler); field reference (BLISS)
W	word integer
X	context dependent (generic)
Y	context dependent (generic)
Z	unspecified or non-standard

N, P, and T strings are typically variable length. Frequently in structures or I/O records they contain a byte-sized digit or character count preceding the string. If so, the location or offset is to the count. Counted strings cannot be passed in CALLs. Instead, a string descriptor is generated.

* - The letters A, C, G, H, and U should be used in preference to L, B, L, W, and B respectively when transportability is involved. The following table defines their sizes:

letter	16	32	36
A	16	32	18
C	8	8	7
G	16	32	36
H	16	16	18
U	8	8	36

12.3 FACILITY PREFIX TABLE

Following is a list of all the facility prefixes. This list will grow over time as new facility prefixes are chosen. No one should use a new code without first "signing out" the prefix with the author of this chapter. Each facility has a typical style of interface, see the Functional and Interface Specifications chapter, and a condition value<31:16> code.

prefix	facility	interface type (see Chap 13)	condition <31:16>
BAS	BASIC support library	V	26
B32	BLISS-32 support library	V	27
BLI	BLISS transportable support library	V	20
CH	Character handling (BLISS)	-	-
CHF	Condition Handling Facility arguments	-	-
CME	Compatibility mode emulator	J	??
COB	COBOL support library	V	25
DEB	Debugger	V	??
FAB	RMS File Access Block	-	-
FOR	Fortran support library	V	24
IO	Input/Output functions	-	-
LIB	Miscellaneous routines	any	21
MTH	Math library	F	22
NAM	RMS Name Block	-	-
NET	Network ACP	J	??
OTS	Common Object Time System	V	23
PLI	PL/1 support library	?	??
PR	Processor Registers	-	-
PRV	Privileges	-	-
PSL	Program Status Longword fields	-	-
RAB	RMS Record Access Block	-	-
RM	RMS internals and status codes	V	1
RMS	Record Management System	V	-
SRM	System Reference Manual Misc. offsets	-	-
SS	System Service Status Codes	-	0
SYS	System Services	V	-
TST	Test packages	any	-
XAB	RMS Extra information Access Block	-	-

Individual products such as compilers also get unique facility codes formed from the product name. They must be signed out in the above list. Facility prefixes should be chosen to avoid conflict with file types.

Structure name prefixes are typically local to a facility. Refer to the individual facility documentation for its structure name prefixes. This does not cause problems since these names are not global, so are not known to the linker. They become known at assembly or compile time only by invoking the structure's definition macro explicitly.

[End of Chapter 12]

Title: VAX-11 Software Eng. Interface Specifications -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE13R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: Part of OTS design chapter 2

Author: P. Conklin, T. Hastings

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, I. Nassi, M. Spier, D. Tolman

Abstract: Chapter 13 describes the standards and conventions used by all modules in the VAX-11 Procedure Library, including the Object Time System. The necessary standards are specified to permit many different individuals to contribute modules independently to the VAX-11 library with a consistent interface documentation. To achieve these modularity objectives, this chapter also standardizes the way arguments are passed, and in particular, the way in which strings are returned. It describes a language independent notation for procedure parameters, including the type of access, the data type, the argument passing mechanism, and the form of the argument.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	T. Hastings	17-Jan-77
Rev 2	Revised from Review	T. Hastings	21-Jan-77
Rev 3	Integrated with Soft Eng Manual	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Extracted procedure specification notation from OTS chapter 2 (PL2R1).
2. Added routine interface types including applications languages.
3. Remove descriptor code numbers and redundant alphabectical table.
4. Quad by-value is ok only on function values.
5. Note op sys option on s descriptor.
6. Change <data type> U to Z for compatibility.
7. Note length and string descriptor pair is length first.
8. Clarify that data type is always the ultimate use.
9. Allow references only to data size.
10. Add data types c, u, h, g.
11. Drop label arg form.
12. Add summary table.
13. Add data type cp.
14. Allow value args to be less than longword in reference use. (Allocation is still longword.)

Rev 1 to Rev 2:

1. Add <data type> codes la, las, and lc.
2. Add <arg form> code d.
3. Add braces notation for repeated arguments.
4. Add = notation for default value
5. Add <arg form> code p.
6. Clarify use of <data type> with call by value <arg mechanism> for other than 32 bits.
7. Change <data type> C to T for compatibility.

8. Change <data type> la to a for compatibility.

9. Change <data type> las to arl, arw, arb.

[End of SE13R3.RNO]

CHAPTER 13

FUNCTIONAL AND INTERFACE SPECIFICATIONS

28-Feb-77 -- Rev 3

This chapter describes the standards and conventions used by all modules in the VAX-11 Procedure Library, including the Object Time System. The necessary standards are specified to permit many different individuals to contribute modules independently to the VAX-11 library with a consistent interface documentation. To achieve these modularity objectives, this chapter also standardizes the way arguments are passed, and in particular, the way in which strings are returned. It describes a language independent notation for procedure parameters, including the type of access, the data type, the argument passing mechanism, and the form of the argument.

The VAX-11 Procedure Library is a collection of routines that provide various services to the calling program. It is made up of a number of sub-libraries. The Math library contains all those functions that perform the traditional Fortran mathematical functions. The common Object Time System is a collection of resource and environment control routines that are common to all application language environments. Each compiler has a library of routines for which it implicitly generates code. Finally, the general library contains routines that are of general use and typically would be called explicitly by the programmer.

13.1 ROUTINE INTERFACE TYPES

In order to achieve the VAX-11 goal of being able to mix languages within a program, all routines are designed with certain attributes in common. The data types and mechanism passing rules are constrained to maximize the ability to interface to routines. A common notation is used to express the specification of the interface.

The access types, data types, mechanisms, and argument forms are defined in the VAX-11 System Reference Manual. Section 2 of this chapter lists them and gives the procedure interface notation for them. In the design of a procedure interface, in addition to the data types that must be designed, four other choices are important.

1. Whether the routine is CALLed or has a non-CALL interface.

2. Whether its scalar input arguments are by value or by reference.

3. How output strings are returned; this is discussed in the next paragraph.

4. Whether the routine has a function value and whether the value is a status code or a scalar result.

Within any given facility, it is generally preferable to have only one style of these interface choices. The facility table in the Naming Conventions chapter indicates what the conventional interface is for each facility. These are defined below. Other combinations can be chosen but the prospect of user confusion must be traded off against the possible inefficiency of forced consistency.

Output strings can be returned by one of four methods.

- The simplest is for the caller to allocate a fixed length string buffer and pass a descriptor of it. The callee writes the result to this buffer with blank fill.

- The next most general is for the caller to allocate a fixed length string buffer that can hold the maximum length result. The caller passes two arguments, one is the address of where to write the actual length and the other is a descriptor to the buffer. By convention, these two arguments are always adjacent in the argument list with the length first.

- The third mechanism is to pass a varying string descriptor. In this case, the caller allocates a maximum buffer and passes a descriptor that contains fields for both the maximum length and the actual length. The callee updates the actual length field in the descriptor.

- The fourth method is for the caller to pass a dynamic string descriptor. In this case the callee allocates the string buffer and places both the address and the length into the dynamic descriptor.

The choice between these methods is a function of what environmental assumptions can be made in the design of the procedure. For the fixed length method, no assumptions are made. The others all assume that the calling language can support variable length strings or substrings. The dual argument form can be used without requiring variable length strings, but gives most of the advantages of them to languages that support them. The varying and dynamic schemes both require languages that support varying length strings. Furthermore, the dynamic method requires the support of a dynamic storage management system.

The most common combinations of interface specifications are given in the following table. The column "scalars" shows how scalars are passed. The column "strings" shows how output strings are returned. The column "function" shows what kind of function value is returned.

type of call	instr- uction	passing scalars	output strings	function value
J (non-CALL)	JSB	parameter	-	-
V (by Value)	CALL	AP by value	length,descr	.lc
F (Function)	CALL	AP by reference	none	scalar
Fortran	CALL	AP by reference	fixed	any
COBOL	CALL	AP by reference	fixed	none
BASIC	CALL	AP by reference	dynamic	any

13.2 NOTATION FOR DESCRIBING PROCEDURE ARGUMENTS

A concise language-independent notation is used to describe each argument to a library procedure. It is suggested that this notation be used for documenting all procedures in the procedure library and in the procedure header itself under CALLING SEQUENCE or FORMAL PARAMETERS. The notation is a compatible extension to the one used in the VAX-11 System Reference Manual. However, the goal of the notation is to describe the formal parameter specified by each list entry in a language independent way. The System Reference Manual only describes the immediate operand specifier, rather than the argument being pointed to. Therefore, additional qualifiers have been added to the System Reference Manual notation. Note that if a parameter is an address which is saved for later access by another procedure, the notation should reflect the ultimate access to be made by the second procedure.

The notation specifies for each argument:

1. A mnemonic name

2. The type of access the procedure will make (read, write,...)

3. The data type of the argument (longword, floating,...)

4. The argument passing mechanism (value, reference, descriptor)

5. The form of the argument (scalar, array,...)

13.2.1 Procedure Parameter Qualifiers

Subroutines are described as:

```
CALL subroutine_name(arg1, arg2, ..., argn)
```

and functions are described as:

```
function_value = function_name(arg1, arg2, ..., argn)
```

where argi and function_value are:

```
<name>.<access type><data type>.<arg mechanism><arg form>
```

where:

1. <name> is a mnemonic for the procedure formal specifier or function value specifier.

2. <access type> is a single letter denoting the type of access that the procedure will (or may) make to the argument:

   r - argument may be read only

   m - argument may be modified, i.e., read and written.

   w - argument may be written only.

   j - argument is an address to be (optionally) JMPed to after stack unwind (return). No <data type> field is given since the argument is a sequence of instructions, e.g., Fortran ERR=.

   c - argument is an address of a procedure to be (optionally) CALLed after stack unwound (return). No <data type> field is given since the argument is a sequence of instructions.

   s - argument is an address of a procedure subroutine to be (optionally) CALLed without unwinding the stack. No <data type> field is given since the argument is a sequence of instructions.

   f - argument is an address of a function to be (optionally) CALLed without unwinding the stack. The <data type> field indicates the data type of the function value.

   a - reserved for use in the System Reference Manual (address). Not used here since the object pointed to is specified.

   b - reserved for use in the System Reference Manual (branch destination). Not used here since a branch destination cannot be a procedure formal.

   v - reserved for use in the System Reference Manual (variable bit field).

3. <data type> is a letter denoting the primary data type with trailing qualifier letters to further identify the data type. Note that the routine must reference only the size specified to avoid improper access violations.

Letters	Use
z	Unspecified
v	Bit (variable bit field)
bu	Byte Logical (unsigned)
c	Single character
u	Smallest unit for addressable storage
wu	Word Logical (unsigned)
lu	Longword Logical (unsigned)
a	Absolute virtual address
cp	Character pointer
lc	Longword containing a completion code
qu	Quadword Logical (unsigned)
b	Byte Integer (signed)
arb	Byte containing a relative virtual address (*)
w	Word Integer (signed)
h	Integer value for counters
arw	Word containing a relative virtual address (*)
l	Longword Integer (signed)
g	General value
arl	Longword containing a relative virtual address (*)
q	Quadword Integer (signed)
f	Single-Precision Floating
d	Double-Precision Floating
fc	Complex (Floating)
dc	Double-Precision Complex
t	text (character) string
nu	Numeric string, unsigned
nl	Numeric string, left separate sign
nlo	Numeric string, left overpunched sign
nr	Numeric string, right separate sign
nro	Numeric string, right overpunched sign
nz	Numeric string, zoned sign
p	Packed decimal string
x	Data type indicated in descriptor

* - arl, arw, and arb is a self-relative address using the same format as the hardware displacements. That is the self-relative address is a signed offset in bytes with respect to the first byte following the argument.

4. <arg mechanism> is a single letter indicating the argument mechanism that the called routine expects:

   v - value, i.e., call-by-value where the contents of the argument list entry is itself the argument of the indicated data type. Note: Call-by-value argument list entries are always allocated as a longword. The quadword data types can be used as values only for function values, never as a formal parameter. Note: the VAX-11 calling standard requires that <access type> must be r whenever <arg mechanism> is v, except for function values where <access type> is always w and <arg mechanism> is usually v.

   r - reference, i.e., call-by-reference where the contents of the argument list entry is the longword address of the argument of the indicated data type. If the argument is a scalar of the indicated data type or is a label, <arg form> must be absent. If the argument is an array, <arg form> must be present.

   d - descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a descriptor. The descriptor is two or more longwords that specify further information about the argument, see the System Reference Manual Appendix C. Note: when <arg mechanism> is d, <arg form> must be present to indicate the type of descriptor.

5. <arg form> is a letter denoting the form of the argument:

   Null means scalar of indicated data type.

   a - array reference or array descriptor, i.e., call-by-reference or call-by-descriptor as indicated by <arg mechanism>. For array call-by-reference the contents of the argument list entry is the address of an array of items of the indicated data type. The length is fixed, implied by entries in the array, e.g., a control block, determined by another argument, or specified by prior agreement. For array call-by-descriptor, the contents of the argument list entry is the longword address of an array descriptor block see the System Reference Manual Appendix C.

   s - string descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a two longword string descriptor. The descriptor contains the length, data type, and address of the string. When the string is written neither the length nor the address fields in the descriptor are modified and the string is filled with trailing spaces or a separate argument is updated with the written length.

v - varying string descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a three longword string descriptor. The descriptor contains length, data type, address, and maximum length. See Appendix C of the System Reference Manual. When the string is written, the length field of the descriptor is also modified but the address and maximum length fields are unaltered.

d - dynamic string descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a two longword string descriptor of the same format as s. However, when the string is written, both the length and address fields may be modified. Space is allocated dynamıcally by routines in the procedure library and is garbage collected periodically

p - Procedure descriptor, i.e., call-by-descriptor where the contents of the argument list entry is the longword address of a two longword procedure descriptor. The descriptor contains the address of the procedure and the data type that the procedure returns if it is a function. <access type> must be c, f, j, or s.

13.2.2 Optional Arguments And Default Values

Optional arguments are enclosed in square brackets, e.g. CALL FOR$READ_SU (unit.rb.v [,err.j.rl [,end.j.rl]]). The caller may omit optional parameters at the end of a parameter list by passing a shortened list. The caller may omit optional parameters anywhere by passing a 0 value as the contents of the argument list entry. A caller may not omit a parameter that is not indicated as optional. The called procedure is not obligated to detect such a programming error. An equal sign (=) after an argument inside square brackets indicates the default value if the argument is omitted. For example, success.wlc.v = SYS$DELLOG (lognam.rt.ds [,tblflg.rb.v=0]).

13.2.3 Repeated Arguments

Arguments or pairs of arguments that may be repeated one or more times are indicated inside braces, e.g. CALL FOR$OPEN ({keywd.rw.v, info.rl.v}). Repeated arguments that may be omitted entirely are indicated inside braces inside square brackets, e.g. CALL FOR$CLOSE ([{logical_unit.rl.v}]).

13.2.4 Examples

```
Sine_of angle.wf.v = MTH$SIN (angle_in_radians.rf.r)

CALL FOR$READ_SF (unit.rb.v, format.mbu.ra [,err.j.rl [,end.j.rl]])
```

Note: That (1) end may be omitted, (2) err and end may both be omitted. However, unit and format must always be present. The argument count byte in the argument list specifies how many arguments are present. Alternatively err, end, or both could have a 0 argument list entry in the above.

Common combinations are:

Completion code:	Status.wlc.v =...
longword call-by-value input arg:	no of pages.rlu.v
address of an array of signed words for input:	array.rw.ra
address of a control block:	fab.mz.ra
address of a precompiled format statement:	format.rbu.ra
label to jump to:	error_label.j.r
floating input call-by-reference arg:	angle_in_rad.rf.r
floating complex call-by-reference input arg:	angle.rfc.r
read only Fortran character string:	string rt.ds
BASIC character string to be written:	string.wt.dd

13.2.5 Summary Chart Of Notation

<name>.<access type><data type>.<arg mechanism><arg form>

<access type>	
r	Read
m	Modify
w	Write
j	RET and JMP
c	RET and CALL
s	sub CALL
f	function CALL

<data type>	
z	Unspecified
v	Bit (variable bit field)
bu	Byte Logical (unsigned)
c	Single character
u	Smallest unit for addressable storage
wu	Word Logical (unsigned)
lu	Longword Logical (unsigned)
a	Absolute virtual address
cp	Character Pointer
lc	Longword containing a completion code
qu	Quadword Logical (unsigned)
b	Byte Integer (signed)
arb	Byte-sized relative virtual address
w	Word Integer (signed)
h	Integer value for counters
arw	Word-sized relative virtual address
l	Longword Integer (signed)
g	General value
arl	Longword-sized relative virtual address
q	Quadword Integer (signed)
f	Single-Precision Floating
d	Double-Precision Floating
fc	Complex (Floating)
dc	Double-Precision Complex
t	text (character) string
nu	Numeric string, unsigned
nl	Numeric string, left separate sign
nlo	Numeric string, left overpunched sign
nr	Numeric string, right separate sign
nro	Numeric string, right overpunched sign
nz	Numeric string, zoned sign
p	Packed decimal string
x	Data type indicated in descriptor

<arg mechanism>	
v	Value
r	Reference
d	Descriptor

<arg form>	
<null>	scalar
a	array
s	fixed string
v	varying length string
d	dynamic string
p	procedure

[End of Chapter 13]

TITLE: BLISS Transportability Guidlines -- Rev 3

Specification Status: draft

Architectural status: Under ECO Control

File: SE14F3.FNO

PDM #: not used

Date: 21-Feb-77

Superseded Specs: none

Author(s): P. Marks, R. Murray, I. Nassi

Typist; G. Hesley, R. Murray

Reviewer(s): R. Brender, D. Cutler, P. Conklin, T. Hastings, S. Hawkinson, D. Tolman, R. Winslow

Abstract: Chapter 14 addresses the process of writing transportable BLISS programs. Tools and techniques are discussed in detail.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	I. Nassi	22-Dec-76
Rev 2	Skipped	I. Nassi	25-Jan-77
Rev 3	SEM Integration	R. Murray	21-Feb-77

Rev 2 to Rev 3

1. Software Engineering Manual integration, this document added as a chapter

Rev 1 to Rev 2

1. revision 2 skipped to align revision histories on all chapters to Rev 3

[end of SE14R3.RNO]

CHAPTER 14

BLISS TRANSPORTABIBILTIY GUIDLINES

21-FEB-77 -- Rev 3

This Chapter addresses the task of writing transportable programs. It is shown that the writing of such code is much easier if considered from the beginning of the project. The properties which cause a program to loose transportablilty are explored. Techniques by which the programmer may avoid these pitfalls are discussed.

14.1 INTRODUCTION

14.1.1 Purpose And Goals

The purpose of this document is to facilitate the process of writing transportable BLISS programs, that is, BLISS programs intended to be executed on architecturally different machines. There are various kinds of solutions to the problem of transportability, each requiring different levels of effort. We feel free in recommending various kinds of solutions. When program text should be rewritten, for example, we suggest doing so. However, it is our belief that large portions of programs can be written which will require absolutely no modification in order to be functionally equivalent over differing architectures. The levels of solutions we see, in order of decreasing desirability, are:

- o no change is needed to program text - transportability is perfectly straightforward.
- o parameterization solves the transportability problem - the program makes use of some features that have an analog on all the other architectures.
- o parallel definitions are required - either programs make use of features of an architecture that do not have analogs across all other architectures, or different, separately transportable aspects of a program interact in non-transportable ways.

The goal is to make transportability as painless as possible, which means that the effort needed in transporting programs should be minimized.

Central to the ideas presented here is the notion that transportability is more easily accomplished if considered from the beginning. Transporting programs after they are running becomes a much more complex task. We suggest frequently running parallel compilations, for instance. It is fortunate therefore, that with the right tools and techniques, transportability is not difficult to achieve. We would also like to point out that the first program is the hardest. Before undertaking a large programming project, we suggest writing and transporting a less ambitious program.

These guidelines are the result of a concentrated study of the problems associated with transportability. We make no claim that these guidelines are complete. We do claim that some of what is contained here will be non-obvious to programmers. We have attempted to identify those areas which, if the programmer is not forewarned, will cause problems. We will be suggesting solutions to all identified problems.

Many of the problems that are discussed here have solutions that are currently being incorporated into the BLISS language, so another way of viewing this document is as a partial rationale for some of these language changes, and a rationale for the definition of BLISS-16 and BLISS-36.

14.1.2 Organization

These guidelines are organized into three sections. The section on General Strategies discusses some high level approaches to writing transportable BLISS software. The section on Tools describes various features of the BLISS language that can be used in solving transportability problems. The section on Techniques analyzes various transportability problems and suggests solutions to them.

14.2 GENERAL STRATEGIES

14.2.1 Introduction

This section presents certain gross or global considerations that are important to the writing of transportable BLISS programs, namely:

- o Isolation, and
- o Simplicity

14.2.2 Isolation

The following maxim should be kept in mind when you are designing and/or coding a program that is to be transported:

- o If it is NON-transportable, isolate it.

You will probably encounter situations for which it is desirable to use machine-specific constructs in your BLISS program. In these cases, simply isolating the constructs will facilitate any future movement of the program to a different machine.

In most cases, only a small percentage of the program or system will be sensitive to the machine on which it is running. By isolating those sections of a program or a system, the effort involved in transporting the program will be confined mainly to these easily identifiable, machine-specific sections.

Specifically, follow these rules:

- o If machine-specific data is to be allocated - place the allocation in a separate MODULE or in a REQUIRE file.
- o If machine-specific data is to be accessed - place the access in a ROUTINE or in a MACRO and then place the ROUTINE or MACRO in a separate MODULE or in a REQUIRE file.
- o If a machine-specific function or instruction is to be used, isolate it by placing it too in a REQUIRE file.
- o If it is impossible or impractical to isolate this part of your program from its module, comment it heavily. Make it very obvious to the reader that this code is non-transportable.

The above rules are applicable in the local context of a routine or module. In a larger or more global context (fo: instance, in the design of an entire system) isolation is implemented by the technique of modularization.

By separating those parts of the system which are machine or operating system dependent from the rest of the system, the task of transporting the entire system is simplified. It becomes a matter of recoding a small section of the total system. The major portion of the code (if written in a transportable manner) should easily make the move to a new machine with a minimum of re-coding effort.

BLISS is a language which facilitates both the design and programming of programs and systems in a modular fashion. This feature should be taken advantage of when writing a transportable system.

14.2.3 Simplicity

A basic concept in writing transportable BLISS software is simplicity - simplicity in the use of the language.

BLISS was originally developed for the implementation of systems software. As a result of this, BLISS is nearly unique among high-level programming languages in that it allows ready access to the machine on which the program will be running. The programmer is allowed to have complete control over the allocation of data, for example.

The same language features that allow access to underlying features of the hardware are very often used to excess. In order to identify those features of the language causing a program to be non-transportable, it is often the case that such features be invoked explicitly, making the program inherently more complex. Reducing the complexity of data allocation, for example, results in a transportable subset of the BLISS language. This reduction of complexity is one of the basic themes that runs through the guidelines.

In effect, the coding of transportable programs is a simpler task because the number of options available has been reduced. Simplicity in the coding effort is one of the reasons for the development of higher-level languages like BLISS. The use of the defaults in BLISS will result in programs which are much more easily transported.

14.3 TOOLS

This section on tools presents various language features that provide a means for writing transportable programs. These features are either intrinsic to BLISS or have been specifically designed for transportability/software engineering uses.

The tools described here will be used throughout the companion section on techniques.

14.3.1 Literals

Literals provide a means for associating a name with a compile-time constant expression. In this section, we will consider some built-in literals which will aid us in writing transportable programs. In addition, we will discuss restrictions on user-defined literals.

14.3.2 Predeclared Literals

One of the key techniques in writing transportable programs is parameterization. Literals are a primary parameterization tool. The BLISS language has a set of predeclared, machine specific literals that can be most useful.

These literals parameterize certain architectural values of the three machines. The values of the literals are dependent on the machine that the program is currently being compiled for. Here are their names and values:

Description	Literal Name	10/20	VAX-11	11
Bits per addressable unit	%BPUNIT	36	8	8
Bits per address value	%BPADDR	18	32	16
Bits per BLISS value	%BPVAL	36	32	16
Units per BLISS value	%UPVAL	1	4	2

The names beginning with '%' are the literal names that can be used. These literal names will be used throughout the guidelines.

Bits per value is the maximum number of bits in a BLISS value. Bits per unit is the number of bits in the smallest unit of storage that can have an address. Bits per address refers to the maximum number of bits an address value can have. Units per value is the quotient %BPVAL/%BPUNIT. It is the maximum number of addressable units associated with a value.

We can derive other useful values from these built-in literals. For example:

```
     LITERAL
             HALF_VALUE = %BPVAL / 2;
```

defines the number of bits in half a word (half a longword on VAX-11).

14.3.2.1 User Defined Literals -

A literal is not strictly speaking a self-defining term. The value and restrictions associated with a literal are arrived at by assigning certain semantics to its source program representation. It is convenient to define the value of a literal as a function of the characteristics of a particular architecture, which means that there are certain architectural dependencies inherent in the use of literals.

Because the size of a BLISS value determines the value and/or the representation of a literal, there are some transportability considerations. BLISS value (machine word) sizes are different on each of the three machines. On VAX-11, the size is 32 bits; on the 10/20 systems, it is 36; and the 11 value is 16.

There are two types of BLISS literals: numeric-literals and string-literals. The values of numeric-literals are constrained by the machine word size. The ranges of values for a signed number, i, are:

VAX-11: $-(2^{**}31) \le i \le (2^{**}31) - 1$

10/20: $-(2^{**}35) \le i \le (2^{**}35) - 1$

11: $-(2^{**}15) \le i \le (2^{**}15) - 1$

ALL: $-(2^{**}(\%BPVAL-1)) \le i \le (2^{**}(\%BPVAL-1))-1$

Double precision floating point numbers (%D'number' in BLISS-32) are not supported in BLISS-36 or in BLISS-16.

A numeric literal, %C'single-character', has been implemented. Its value is the ASCII code corresponding to the character in quotes and when stored, it is right-justified in a BLISS value (word or longword). A more thorough discussion of its usage can be found in the section entitled: "Data: Character Sequences".

There are two ways of using string-literals: as integer-values and as character strings. When string-literals are used as values, they are not transportable. This arises out of the representational differences and from differing word sizes. The following table illustrates these potential differences for an %ASCII type string literal:

	VAX-11	10/20	11
Maximum number of characters.	4	5	2
Character placement.	right to left	left to right	right to left

This type of string literal usage and also its use as a character string are discussed in the section entitled: "Data: Character Sequences".

14.3.3 MACROS

BLISS macros can be an essential tool in the development of transportable programs. Because they evaluate (expand) during compilation, it is possible to tailor a program to a specific machine.

A good example can be found in the section on structures. There, two macros are developed which are completely transportable. The macros can determine the number of addressable units needed for a vector of elements, where the element size is specified in terms of bits.

There are also pre-defined machine conditionalization macros available. These macros can be used to compile selectively only certain declarations and/or expressions depending on which compiler is being run.

Their definitions for the bliss-32 set are:

```
MACRO
        %BLISS16[] = % ,
        %BLISS36[] = % ,
        %BLISS32[] = %REMAINING % ;
```

There are analogous definitions for the other machines. The net effect is that in the BLISS-32 compiler, the arguments to %BLISS16 and %BLISS36 will disappear, while arguments to %BLISS32 will be replaced by the text given in the argument list.

14.3.4 Module Switches

A module switch and a corresponding on-off switch are provided to aid in the writing of transportable programs. This switch, LANGUAGE, is provided for two reasons:

- To indicate the intended transportability goals of a module and
- To provide diagnostic checking of the use of certain language features.

The programmer can therefore indicate the target architectures (environments) for which a program is intended.

Diagnostic checking consists of the compiler determining whether certain language features are available for all of the intended target environments.

The LANGUAGE switch may be used in the module header or switches declaration to designate which of the several BLISS processors are intended to compile the module.

The syntax is:

```
        LANGUAGE (language-type ,...)
```

where language-type is any combination of BLISS36, BLISS16 or BLISS32.

If no LANGUAGE switch is specified, the default is all three languages, and as a consequence, only the most restricted language facilities are made available.

Each compiler will give a warning diagnostic if its own language is not in the list of language-types.

Within the scope of a language switch, each compiler will give a warning diagnostic for any language construct which is not in the intersection of the specified set of languages.

NOTE

As of this writing the particular language features that will be subject to diagnosis have yet to be detailed. However, using it now will serve to document the program, and to make the program immune to compiler enhancements that restrict certain features under certain switch settings.

Here is an example of how the LANGUAGE switch would be used:

```
        MODULE FOO(...,LANGUAGE(BLISS36, BLISS16, BLISS32),...) =
        BEGIN

                        ...
                        ...
                        ...

        BEGIN

        !+
        ! BLISS16 no longer in effect.
        !-

        SWITCHES
                LANGUAGE(BLISS36, BLISS32);
                        ...
                        ...
                        ...

                Any use of language features, within
                this block, which are specific to
                BLISS16 will result in a diagnostic
                warning.
                The compilation of this section
                of code by a BLISS-16 compiler will
                result in a diagnostic warning.

                        ...
                        ...
                        ...

        END;

        !+
        !  All three language settings are restored.
        !-
```

14.3.5 Reserved Names

The following page contains a list of BLISS reserved names. The list represents the union of reserved names in all three BLISS dialects. Hence, if one is writing a transportable program, one should avoid using any of these names as a user-defined name, since such use results in a compiler diagnostic. Items marekd with an asterisk should not be used when writing code intended to be transportable.

*ADDRESSING_MODE
*ALIGN
ALWAYS
AND
BEGIN
BIND
BIT
*BUILTIN
BY
*BYTE
CASE
CODECOMMENT
COMPILETIME
DECR
DECRA
*DECRU
DO
ELSE
ELUDOM
ENABLE
END
EQL
EQLA
*EQLU
EQV
EXITLOOP
EXTERNAL
FIELD
FORWARD
FROM
GEQ
GEQA
*GEQU
GLOBAL
GTR
GTRA
*GTRU

IF
INCR
INCRA
*INCRU
INITIAL
INRANGE
KEYWORDMACRO
LABEL
LEAVE
LEQ
LEQA
*LEQU
LIBRARY
LINKAGE
LITERAL
LOCAL
*LONG
LSS
LSSA
*LSSU
MACRO
MAP
MOD
MODULE
NEQ
NEQA
*NEQU
NOT
NOVALUE
OF
OR
OTHERWISE
OUTRANGE
OWN
PLIT
PRESET
*PSECT

RECORD
REF
REGISTER
REP
REQUIRE
RETURN
ROUTINE
SELECT
SELECTA
SELECTONE
SELECTONEA
*SELECTONEU
*SELECTA
SET
*SHOW
*SIGNED
STACKLOCAL
STRUCTURE
SWITCHES
TES
THEN
TO
UNDECLARE
UNTIL
UPLIT
*VOLATILE
*WEAK
WHILE
WITH
*WORD
XOR

14.3.6 REQUIRE Files

REQUIRE files are a way of gathering machine specific declarations and/or expressions together in one place.

In many cases, it will be either impossible or unnecessary to code a particular BLISS construct (e.g. routines, data declarations, etc.) in a transportable manner. Developing parallel REQUIRE files, one for each machine, can often provide a solution to transporting these constructs.

For example, if a certain set of routines are very machine specific, then the solution may be to code two or three functionally equivalent routines, one for each machine type, and segregate them each in their own REQUIRE file.

Each BLISS compiler has a pre-defined search rule for REQUIRE file names based on their file types. Each compiler will search first for a file with a specific file type, then it will search for a file with the file type '.BLI'.

The search rules for each compiler are:

Compiler	1st	2nd
BLIS36	.B36	.BLI
BLIS16	.B16	.BLI
BLIS32	.B32	.BLI

Hence, the following REQUIRE declaration:

```
REQUIRE
        'IOPACK';                ! I/O Package
```

will search for IOPACK.B36, IOPACK.B16 or IOPACK.B32, depending on which compiler is being run. Failing that it will look for IOPACK.BLI.

Inherent in these search rules is a naming convention for REQUIRE files. If the file is transportable, give it the file type '.BLI'. If it is specific to a particular dialect, give it the corresponding file type (e.g. '.B36').

14.3.7 ROUTINES

The key to transportability is the ability to identify properties of an environment, abstract the property by giving it a name, and then define the semantics of the property in all applicable environments. The closed subroutine has long been regarded as the principal abstraction mechanism in programming languages. With BLISS, we see other abstraction mechanisms being used, like structures, macros, literals, require files, etc., but the routine can still be easily used as a transportability abstraction mechanism.

For instance, when designing a system of transportable modules which uses the concept of floating point numbers and associated operations, there will be a need to perform floating point arithmetic. The question naturally arises as to the environment in which the arithmetic should be done. If the floating point arithmetic resides entirely in a well-defined set of routines, and no knowledge of the various representations of floating point numbers is used except through these well defined interface routines, then it becomes possible to perform "cross-arithmetic", which becomes highly desirable when writing cross-compilers, for instance. Even if the ability to perform cross-arithmetic is not desired, isolating floating point operations in routines is a good idea since these routines can then be reused more easily in another project. A little thought will indicate that the floating point routines themselves have to be transportable if they are going to perform cross-arithmetic, but need not be transportable if cross arithmetic is a non-goal.

The principal objection to using routines as an abstraction mechanism is that the cost of calling a procedure is non-trivial, and that cost is strictly program overhead. Composing this sort of abstraction in the limit will produce serious performance degradation. For this reason, a programmer should probably try not to use the routine as an abstraction mechanism if a small amount of forethought will be sufficient to enable the writing of a single transportable module.

14.4 TECHNIQUES

This section on techniques shows you how to write transportable programs. The section is organized in dictionary form by BLISS construct or concept. Each sub-section contains:

- A discussion of the construct or concept.
- Transportability problems that its use may engender.
- Specific guidelines and restrictions on the use of the construct or concept.
- Examples - both transportable and non-transportable.

The examples, in all cases, attempt to use the tools described in the TOOLS section.

14.4.1 Data

14.4.1.1 Introduction -

This section deals with the allocation of data in a BLISS program. For the purposes of this section we do not deal with character sequence (string) data or address data. These types of data are discussed in their own sections (See: "Data: Addresses and Address Calculation" and "Data: Character Sequences"). Primarily, we discuss the allocation of scalar data (e.g. counters, integers, pointers, addresses, etc.) A presentation of more complex forms of data can be found in the sections entitled: "Structures and Field-Selectors" and "PLITs and Initialization". First there is a discussion of transportability problems encountered due to differing machine architectures. Next a discussion of the BLISS allocation-unit attribute is presented. Finally, a discussion of other BLISS data attributes that must be considered when writing transportable programs is discussed.

14.4.1.2 Problem Genesis -

The allocation of data (via the OWN, LOCAL, GLOBAL, etc. declarations) tends to be one of the most sensitive areas of a BLISS program in terms of transportability. This problem of transporting data arises chiefly from two sources:

- o The machine architectures and

- o The flexibility of the BLISS language.

When we are considering writing a BLISS program that will be transported to another machine, we are confronted with the problem of allocating data on (at least two) architecturely different machines.

Although we have already discussed differing word sizes, there are further differences. On the VAX-11 machine data may be fetched in longwords (32 bits), in words (16 bits) and in bytes (8 bits); on the 11, both words and bytes may be fetched. Only 36-bit words on the 10/20 systems may be directly fetched (i.e. without a byte pointer).

If we were writing our program in MACRO-10 or MARS we would not consider these differences to be important - clearly, our assembly language program was not intended to be transportable.

What decisions, however, must the BLISS programmer make in the transportable allocation of data? Need he or she be concerned with how many bits are going to be allocated?

These questions (and their answers) can be complicated by the other chief source of data transportability problems, namely the BLISS language itself.

BLISS is different than many other higher-level languages in that it allows ready access to machine-specific control, particularly in storage allocation. This is fortunate for the programmer who is writing highly machine-specific, efficient software. This programmer needs much more control over exactly how many bits of data will be used. This feature of BLISS, however, can complicate the decisions that need to be made by the BLISS programmer who is writing a transportable program. Does he or she allocate scalars by bytes, or by words, or by longwords?

14.4.1.3 Transportable Declarations -

Consider the following simple example of a data declaration in BLISS-32:

```
        OWN
                PAGE_COUNTER: BYTE;      ! Page counter
```

The programmer has allocated one byte (8 bits) for a variable named PAGE_COUNTER. No matter what his or her intentions were in requesting only one byte of storage, this declaration is non-transportable. The concept of BYTE (in this context) does not exist on the 10/20 systems. In fact, in BLISS-36 the use of the word BYTE results in an error message.

If this declaration had been originally coded as:

```
        OWN
                PAGE_COUNTER;            ! Page counter
```

then this could have been transported to any of the three machines. The functionality (in this case, storing the number of pages) has not been lost. We allowed the BLISS compiler to allocate storage by default by not specifying any allocation-unit in the OWN declaration. In all the BLISS dialects the default size for allocation-unit consists of %BPVAL bits. Thus our first transportable guideline is:

- o Do not use the allocation-unit attribute in a scalar data declaration.

Besides the allocation-unit there are other attributes that may present transportability problems if used. In particular, when allocating data:

o Do not use the following attributes:

Extension (SIGNED and UNSIGNED),
Alignment,
Volatile,
Range,
Weak

which is to say: think twice before you write a declaration. Do you really need to specify any data attributes other than structure attributes?

The Extension-attribute specifies whether the sign bit is to be extended in a fetch of a scalar. This attribute is meaningful only on VAX-11 and is not supported by BLISS-36 or BLISS-16. No sign extension can be performed if the allocation unit is not specified.

The Alignment-attribute tells the compiler at what address boundary a data segment is to start. It is not supported in BLISS-36 or BLISS-16; hence, it is non-transportable. Suitalbe default alignments are available dependent on the size of the scalar.

The Volatile-attribute notifies the compiler that code to fetch the contents of this data segment must be generated anew for each fetch in the BLISS program. It is not supported in BLISS-36 or BLISS-16 and will result in a compiler diagnostic.

The Range-attribute specifies the number of bits needed to represent the value of a literal that is declared global in a separately compiled module. The STARLET linker is the only linker that currently supports external literals.

The Weak-attribute is a STARLET-specific attribute and is not supported by BLISS-36 or BLISS-16. It can not be used in a transportable program.

These guidelines are relatively simple, yet they should relieve the BLISS programmer of needing to worry about how the program data will actually be allocated by the compiler. There is often very little reason to specify an allocation-unit or any attributes. The default values are almost always sufficient.

In the case of scalar data, the use of the default allocation-unit will sometimes result in the allocation of more storage than is strictly necessary. This gain in program data size (which, in most instances, is small) should be weighed against a decrease in fetching time for a particular scalar value, and the knowledge that because of the default alignment rules, no storage savings may, in fact, be realized.

In the BLISS language, the default size of %BPVAL bits was chosen (among other reasons) because this is the largest, most efficiently accessed unit of data for a particular machine. Which is to say, the saving of bits does not necessarily mean a more efficient program.

There will undoubtedly be cases where it is impossible to avoid the use of one or more of the above attributes. In fact, it may be desirable to take advantage of a specific machine feature. In these cases follow this guideline:

- Conditionalize and/or heavily comment the use of declarations which may be non-transportable.

This guideline is the "escape-hatch", if you will, in this set of guidelines. It should only be used sparingly and where justified. To use it often will only result in more code that will need to be re-written when the program has to be transported to another machine - and that's not our goal.

14.4.2 Data: Addresses And Address Calculations

14.4.2.1 Introduction -

This section will discuss address values and calculations using address values. First, there will be a presentation of the problems that might occur when using an address or the result of an address calculation as a value. A transportable solution to some of these problems is then presented. Next, a discussion of the need for address forms of the BLISS relational operators and control expressions and how and when to use them will be presented. Finally, some important differences in the interpretation of address values between BLISS-10 and BLISS-36 are discussed.

14.4.2.2 Addresses And Address Calculations -

The value of an undotted variable name in BLISS is an address. In most cases, this address value is used only for the simple fetching and storing of data. When address values are used for other purposes, we must be concerned with the portability of an address or an address calculation. By address calculation we mean any arithmetic operations performed on address values.

The primary reason for our concern is the different sizes (in bits) of addressable units, addresses, and BLISS values (machine words) on the three machines. For convenience in writing transportable programs, these size values have been parameterized and are now predeclared literals. A table of their values can be found in the section entitled: "Literals".

To see how these size differences can have an effect on writing transportable programs, let's consider a common type of address expression; namely an expression that computes an address value from a base (a pointer or an address) and an offset. That is, some expression of the form:

```
                    ... base + index ...
```

Now consider the following BLISS assignment expression using this form of address calculation:

```
          OWN
                  ELEMENT_2;
          .
          .
          .

          ELEMENT_2 = .(INPUT_RECORD + 1);
```

The intent (most likely) was to access the contents of the second value in the data segment named INPUT_RECORD and to place that value in an area pointed to by ELEMENT_2. The effect, however, is different on each machine as we shall see.

By adding 1 to an address (in this case, INPUT_RECORD) we are computing the address of the next addressable unit on the machine. In BLISS-32 and BLISS-16 this would be the address of the next byte (8 bits), but in BLISS-36 this would be the address of the next word (36 bits). This is probably not a transportable expression because of the different sizes of the addressable units and the resultant values.

Based on the above example, we introduce the following guideline:

- o When a complex address calculation is not an intrinsic part of the algorithm being coded, do not write it outside of a structure declaration.

There is a way, however, of making such an address calculation transportable. It involves the use of the values of the predeclared literals. In the last example, if the index had been 4 in BLISS-32 or 2 in BLISS-16 then in each case we would have accessed the next word.

We need to calculate a multiplier that will have a value of 4 in BLISS-32, 2 in BLISS-16 and 1 in BLISS-36. Such a multiplier already exists as another predeclared literal. Its definition is %BPVAL/%BPUNIT, and it is called %UPVAL.

Using this literal in our example we would have:

```
ELEMENT_2 =
        .(INPUT_RECORD + 1 * %UPVAL);
```

The address expression is now tranportable.

This last example raises an interesting point. If an address calculation of this form is used then it is very likely that the data segment should have had a structure such as a VECTOR, BLOCK or BLOCKVECTOR associated with it. The last example could have then been coded as:

```
OWN
    INPUT_RECORD:
        FLEX_VECTOR[RECORD_SIZE,%BPVAL],
    ELEMENT_2;
.
.
.

ELEMENT_2 = .INPUT_RECORD[1];
```

The transportable structure FLEX_VECTOR and a more thorough discussion of structures can be found in the section entitled: Structures and Field Selectors.

14.4.2.3 Relational Operators And Control Expressions -

The previous example illustrated the use of address values in the context of computations. Other common uses of addresses are in comparisons (testing for equality, etc.) and as indices in loop and select expressions. The use of address values in these contexts points to another set of differences found amongst the three machines.

In BLISS-32 and BLISS-16, addresses occupy a full word (%BPADDR equals %BPVAL) and unsigned integer comparisons must be performed. However, in BLISS-36, addresses are smaller than the machine word (18 versus 36 bits) and signed integer operations are performed for efficiency reasons.

It can be seen that to perform a simple relational test of address values:

```
... ADDRESS_1 LSS ADDRESS_2 ...
```

requires two different interpretations. This expression would evaluate correctly on the 10/20 systems. But, on the VAX-11 and 11 machines, the following would have had to have been coded for the comparison to have been made correctly:

```
... ADDRESS_1 LSSU ADDRESS_2 ...
```

Another type of relational operator, designed specifically for address values, is needed. Such operators exist and are referred to as address-relational-operators. BLISS-36, BLISS-16 and BLISS-32 have, in fact, a full set of them (e.g. LSSA, EQLA, etc.) which support address comparisons.

In BLISS-16 and BLISS-32, the address-relationals are equivalent to the unsigned-relationals. In BLISS-16, the address-relationals are equivalent to the signed-relationals. For all practical cases, a user need not be concerned with this, since this "equivalencing" permits equivalent address comparisons to be performed across architectures.

In addition, there are address forms of the SELECT (SELECTA), SELECTONE (SELECTONEA), INCR (INCRA) and DECR (DECRA) control expressions. The following guidelines establish a usage for these operators and contol expressions:

- o If address values are to be compared, use the address form of the relational operators.
- o If an address is used as an index in a SELECT, SELECTONE, INCR or DECR expression, use the address form of these control expressions.

A violation of either of these guidelines can have unpredictable results.

14.4.2.4 BLISS-10 Addresses Versus BLISS-36 Addresses -

There is a fundamental conceptual change from BLISS-10 to BLISS-36 in the defined value of a name. BLISS-10 defines the value of a data segment name to be a byte pointer consisting of the address value in the low half of a word, and position and size values of 0 and 36 in the high half of the word. BLISS-36, however, defines the value as simply the address in the low half and zeros in the high half. This change was made solely for reasons of transportability, since it allows BLISS to assign uniform semantics to an address.

The fetch and assignment operators are redefined to use only the address part of a value. Thus the expressions:

```
                         Y = .X;
                         Y = F(.Y) + 2;
```

are the same in both BLISS-10 and BLISS-36, but

```
                         Y = X;
```

assigns a different value to Y in BLISS-36 and in BLISS-10.

Field selectors are still available but must be thought of as extended operands to the fetch and assignment operators, instead of as value producing operators applied to a name. Thus the meaning of:

```
                         Y<0,18> = .X<3,7>;
```

is unchanged, but

```
                         Y = X<3,7>;
```

is invalid. Moreover, it is highly recommended that field selectors never appear outside of a structure declaration, since bit position and size are apt to be highly machine dependent. A more thorough discussion can be found in the section entitled: Structures and Field Selectors.

14.4.3 Data: Character Sequences

14.4.3.1 Introduction -

This section will discuss the use of character sequences (strings) in BLISS programs. Historically, there has been no consistent method for dealing with strings and the functions operating upon them. Ad hoc string functions have been the rule, having been implemented by individuals or projects to suit their particular needs. This section will begin by looking at quoted strings in two different contexts. We will discuss transportability problems associated with quoted strings, and guidelines for their use.

Quoted strings are used in two different contexts:

- o as values (integers) and

- o as character strings

14.4.3.2 Usage As Numeric Values -

The use of quoted strings as values (in assignments and comparisons) illustrates the problem of differing representations on differing architectures. Describing the natural translation of a string literal for each architecture will illustrate the problem. For example, consider the following code sequence:

```
OWN
        CHAR_FOO;           ! To hold a literal

CHAR_FOO = 'FOO';
```

A natural interpretation for BLISS-32 to use is that one longword would be allocated and the three characters would be assigned to increasing byte addresses within the longword. In memory, the value of CHAR_FOO would have the following representation:

```
CHAR_FOO:   / 00 O O F / (32)
```

BLISS-16 would not allow this assignment because only two ASCII characters are allowed per string-literal. This restriction arises from the fact that BLISS-16 works with a maximum of 16-bit values and three 8-bit ASCII characters require 24 bits.

On the 10/20 systems a word would be allocated and the characters would be positioned starting at the high-order end of the word. Thus the string-literal would have the following representation in memory:

```
CHAR_FOO:   / F O O 00 00 0 / (36)
```

Even if the 10/20 string-literal had been right-justified in the word, it still would not equal the VAX-11 representation, numerically. So, in fact, the following would not be transportable:

```
WRITE_INTEGER( 'ABC' );
```

since 'ABC' is invalid syntax in BLISS-16, has the value -33543847936 in BLISS-36, and the value 4276803 in BLISS-32.

Based on these problems with representation our first guideline is:

- Do not use string-literals as numeric values.

In those cases where it is necessary to perform a numeric operation (e.g. a comparison) with a character as an argument, you must use the %C form of integer literal. This literal takes one character as its argument and returns as a value the integer index in the collating sequence of the ASCII character set, so that:

```
%C'B' = %X'42' = 66
```

The %C notation was introduced to standardize the interpretation of a quoted string across all possible ASCII-based environments. %C'quoted-character' can be thought of as "right-adjusting" the character in a bit string containing %BPVAL bits.

14.4.3.3 Usage As Character Strings -

The necessity of using more than one character in a literal leads us to the other situation in which quoted strings are used: as character strings.

To facilitate the allocation, comparison and manipulation of character sequences, a built-in character sequence function package has been introduced to the BLISS language. It has been implemented in BLISS-32 and BLISS-36 and plans exist to implement it in BLISS-16.

These built-in functions provide a very complete and powerful set of operations on characters. Our next guideline is:

- o You must use the built-in function package when allocating and operating upon character sequences. This is the only way one can guarantee the portability of strings and string operations.

A more detailed description of these functions can be found in the Character Handling Functions chapter of the BLISS-VAX Language Guide, Second Edition.

14.4.4 PLITs And Initialization

14.4.4.1 Introduction -

This section is primarily concerned with PLITs and their uses. First, there is general discussion of PLITs and the contexts in which they often appear. A presentation of how scalar PLIT items should be used follows. Next, the problems involved in using string literals in PLITs and suggested guidelines for their use are presented. Finally, the use of PLITs to initialize data segments will be illustrated by the development of a transportable table of values.

14.4.4.2 PLITs In General -

Because BLISS values are a maximum of a machine word in length, any literal that requires more than a word for its value needs a separate mechanism, and that mechanism is the PLIT (or UPLIT). Hence, PLITs are a means for defining references to multi-word constants. PLITs are often used to initialize data segments (e.g. tables) and are used to define the arguments for routine calls.

PLITs themselves are transportable; however, their constituent elements and their machine representation are not always transportable.

A PLIT consists of one or more values (PLIT items). PLIT items may be strings, numeric constants, address constants or any combination of these last three, providing that the value of each is known prior to execution time.

14.4.4.3 Scalar PLIT Items -

The first transportability problem that might be encountered with the use of PLITs is in the specification of scalar PLIT items. As with any other declaration of scalar items (pointers, integers, addresses, etc.) it is possible to define them with an allocation-unit attribute. For example, in BLISS-32, we can specify such machine specific sizes as BYTE and LONG. Thus the following example is non-transportable and, in fact, will not compile on BLISS-36 or BLISS-16:

```
        BIND
                Q1 = PLIT BYTE(1, 2, 3, LONG -4);
```

This last example provides the first PLIT guideline:

- o Do not use allocation-units in the specification of a PLIT or PLIT item.

Thus, the BIND should have been coded as follows:

```
        BIND
                Q1 = PLIT(1, 2, 3, -4);
```

This last guideline is necessary because of the differences in the sizes of words on the three machines, a feature of the architectures. A discussion of the role of machine architectures in the transportability of data can be found in the section entitled: "Data". Further guidelines are presented in the section entitled: "Intializing Packed Data".

14.4.4.4 String Literal PLIT Items -

The next guideline is based on the representation of PLITs in memory. Specifically the problem is encountered when scalar and string PLIT items appear in the same PLIT.

The difficulty arises primarily from the representation of characters on the different machines. A more thorough discussion of character representation can be found in the section entitled: "Data: Character Sequences".

Care must be exercised when strings are to be used as items in PLITs. For example, we may wish to specify a PLIT that consists of two elements: a 5-character string and an address of a routine. If we specify it as:

```
        BIND
                CONABC = PLIT('ABCDE', ABC_ROUT);
```

then the VAX-11 representation is as follows:

```
        CONABC:                    / D C B A /  (32)

                                   /       E /  (32)

                                   / address /  (32)
```

on the 11, it would be:

```
        CONABC:                        / B A /  (16)

                                       / D C /  (16)

                                       /   E /  (16)

                                  / address  /  (16)
```

and the 10/20 representation would be:

```
        CONABC:                   / A B C D E / (36)

                                  / address   / (36)
```

The three PLITs are not equivalent. Three longwords are required for the BLISS-32 representation, four words are needed for BLISS-16, and two words are needed for the BLISS-36 representation. If we wished to access the two elements of this PLIT by the use of an address offset, we would have problems. For example, the second element (the address) is accessed by the expression:

```
                    ... CONABC + 1 ...
```

in the BLISS-36 version, but not in the BLISS-32 or BLISS-16 versions. For the BLISS-32 version, we would need the expression:

```
                    ... CONABC + 8 ...
```

and for BLISS-16, it would have to be:

```
                    ... CONABC + 6 ...
```

Taking a data segment's base address and adding to it an offset (as in this case) is particularly sensitive to transportability. A discussion on the use of addresses can be found in the section entitled: "Data: Addresses and Address Calculations".

This section on addresses suggests the use of the literal, %UPVAL, to ensure some degree of transportability. Its value is the number of addressable units per BLISS value (machine word). As already discussed, in BLISS-32, the literal equals 4; in BLISS-16, it is 2; and in BLISS-36, its value is 1.

Multiplying an offset by this value can, in some cases, ensure an address calculation that will be transportable. So to access the second element in the above PLIT, one would write:

```
               ... CONABC + 1*%UPVAL ...
```

But this won't work for the VAX-11 representation. An offset value of 8 is needed because the string occupies two words (BLISS values). The situation is similar for the 11 version, where the string occupies 3 words and would need a offset value of 6 not 2.

The problem with this particular example (and, in general, with strings in PLITs) is not in the use of a string literal but in its position within the PLIT. Because the number of characters that will fit in a BLISS value differs on all three machines (see the section: Data: Character Sequences), the placement of a string in a PLIT will very often result in different displacements for the remaining PLIT items.

There is a relatively simple solution to this problem:

o In a PLIT there can only be a maximum of one string literal, and that literal must be the last item in a PLIT.

Following this guideline, the example should have been coded:

```
        BIND
             CONABC = PLIT( ABC_ROUT, 'ABCDE');
```

and this expression:

```
                    ... CONABC + 1*%UPVAL ...
```

would have resulted in the address of the second element in the PLIT (in this case the string).

14.4.4.5 An Example Of Initialization -

As mentioned in the beginning of this section, PLITs are often used to initialize data segments such as tables. A data segment allocated by an OWN or GLOBAL declaration can be initialized by using the INITIAL attribute. The INITIAL attribute specifies the initial values and consists of a list of PLIT items.

A good example which shows how relatively easy it is to initialize data in a transportable way is to illustrate the process one might use to build a table of employee data. Information on each employee will consist of three elements: an employee number, a cost center number and the employee's name. The employee's name will be a fixed length, 5-character field.

For example, a line of the table would contain the following information:

```
                    345     201         MARKS
```

Converting this line into a list of PLIT items which conform to this section's guidelines would result in the following:

```
                         (345, 201, 'MARKS')
```

Notice that no allocation units were specified and that the character string was specified last. We will now use this line to initialize a small table of only one line. The table will have the built-in BLOCKVECTOR structure attribute. The table declaration would look like:

```
        OWN
                TABLE:
                        BLOCKVECTOR[1,3]
                        INITIAL(
                                345,
                                201,
                                'MARKS'
                                );
```

A problem, however, has developed. This definition would work well in BLISS-36. That is, three words would have been allocated for TABLE. The first word would have been initialized with the employee number; the second word with the cost center; and the third with the name. But the declaration would not be correct in BLISS-32 or BLISS-16, simply because not enough storage would have been allocated for all the initial values. BLISS-32 would have required 4 longwords and the BLISS-16, 5 words.

The problem arises as a result of the way in which strings are represented and allocated on the three machines (see the section: Data: Character Sequences). The solution is simple. We only need to determine the number of BLISS values (words) that will be needed for the character string on each machine. There is a function that will give this value. It is named CH$ALLOCATION and it is part of the Character Sequence Function Package. It takes as an argument the number of characters to be allocated and returns the number of words needed to represent a string of this length. We can use this value as an allocation actual in the table definition, as follows:

```
        OWN
                TABLE:
                        BLOCKVECTOR[1,2 + CH$ALLOCATION(5)]
                        INITIAL(
                                345,
                                201,
                                'MARKS'
                                );
```

The declaration is now transportable. By using the CH$ALLOCATION function we can be assured that enough words will be allocated on each machine. No recoding will be necessary.

We are free to add other lines to the table and not be concerned with the representation or allocation of the data. Here is a larger example of the same kind of table. We won't develop it step by step, but point out and explain some of the highlights.

The example:

```
                ...
                ...
                ...

!+
!       Table Parameters
!-

LITERAL
        NO_EMPLOYEES = 2,
        EMP_NAME_SIZE = 25,
        EMP_LINE_SIZE = 2 +
            CH$ALLOCATION(EMP_NAME_SIZE);

!+
!       Employee Name Padding Macro
!-

MACRO
        NAME_PAD(NAME) =
            NAME, REP CH$ALLOCATION(EMP_NAME_SIZE -
                            %CHARCOUNT(NAME)) OF (0) %;

!+
!       Employee Information Table
!
!       Size: NO_EMPLOYEES * EMP_LINE_SIZE
!-

OWN
        EMP_TABLE:
                BLOCKVECTOR[NO_EMPLOYEES, EMP_LINE_SIZE]
                INITIAL(
                        345,
                        201,
                        NAME_PAD('MARKS PETER'),

                        207,
                        345,
                        NAME_PAD('NASSI ISAAC')

                        );
                ...
                ...
                ...
```

The literals serve to parameterize certain values that are subject to change. The literal EMP_LINE_SIZE has as its value the number of words needed for a table entry. The character sequence function, CH$ALLOCATION, returns the number of words needed for EMP_NAME_SIZE characters.

The macro will, based on the length of the employee name argument (NAME), generate zero-filled words to pad out the name field. Thus, we are assured of the same number of words being initialized for each employee name, no matter what its size might be. This is important because storage is allocated according to the fixed length of a character field (employee name). The actual string length may, of course, be less than that value.

This last example was developed with the specification that the employee name field was fixed in length (EMP_NAME_SIZE). What if, however, we wished to have the table hold variable length names? That is, for certain reasons, we wished to allocate only enough storage to hold the table data, not the maximum amount.

The table structure developed above won't work because it is predicated upon the constant size of the name field. If we were to use variable length character strings, either too much or not enough storage would be allocated. And there would be no consistent way of accessing the employee name (where would the next one start?). We could, if we knew the length of every employee name, determine in advance the number of words needed. But this is not a very practical solution.

One transportable solution is to remove the character string from the table and replace it with a pointer (a word in length) to the string. The Character Package has a function, CH$PTR, which will construct a pointer to a character sequence. As an added benefit, this pointer can be used as an argument to the functions in the Character Package. The cost of this technique is the addition of an extra word (the character sequence pointer) for each table entry.

Here is a typical example, again based on the employee table:

```
                    ...
                    ...
                    ...


    !+
    !       Table Parameters
    !-

    LITERAL
            NO_EMPLOYEES = 2,
            EMP_LINE_SIZE = 3;

    !+
    !       Macro to construct a CS-pointer to employee name
    !-

    MACRO
            NAME_PTR(NAME) =
                CH$PTR(UPLIT( NAME )) %;

    !+
    !       Employee Information Table
    !
    !       Size: NO_EMPLOYEES by EMP_LINE_SIZE
    !-

    OWN
            EMP_TABLE:
                    BLOCKVECTOR[NO_EMPLOYEES, EMP_LINE_SIZE]
                    INITIAL(
                            345,
                            201,
                            NAME_PTR('MARKS PETER'),

                            207,
                            345,
                            NAME_PTR('NASSI ISAAC')

                            );
                    ...
                    ...
                    ...
```

14.4.4.6 Initializing Packed Data -

In this section we will discuss some transportability considerations involved in the initialization of packed data. By packed data, we mean that for data values v1, v2, ..., vn with bit-positions p1, p2, ..., pn and bit-sizes of s1, s2, ..., sn, respectively, the value of the PLIT-item would be represented by the following expression:

```
                v1^p1 OR v2^p2 OR ... OR vn^pn
```

where

```
                max(p1, p2, ..., pn) < %BPVAL

                s1 + s2 + ... + sn < %BPVAL
```

and for all i

```
                -2**si < vi < 2**(si - 1)
```

The OR operator could be replaced by the addition operator (+), but the result would be different if, by accident, there were overlapping values. Notice that the packing of data in a transportable manner is dependent on the value of %BPVAL.

We will illustrate the initialization of packed data by modifying the employee table example that was developed above. When accessing a field within a block, it is a common practice to make each field reference (i.e., offset, position and size) into a macro. So, for example, the field reference macros for the original employee table would look like:

```
        MACRO
                EMP_ID          = 0,0,%BPVAL,0 %,
                EMP_COST_CEN    = 1,0,%BPVAL,0 %,
                EMP_NAME_PTR    = 2,0,%BPVAL,0 %;
```

We can make use of these macros in developing an initialization macro. In essence, we are making use of some already parameterized values. This is another example of how we can use parameterization as one of the key techniques in writing transportable code.

If we knew that the number of bits needed to represent the values of EMP_ID and EMP_COST_CEN would each not exceed 16, we could pack these two fields into one BLISS value in BLISS-32 and BLISS-36. In BLISS-16 the definition of the employee table, as it now stands, would allocate only 16 bits for each field, since %BPVAL equals 16. In BLISS-36, we will choose to use an 18-bit size for these two fields, since we know that both DECsystem-10 and DECsystem-20 hardware have instructions that operate efficently on half-words.

Thus, for BLISS-36 and BLISS-32 the field reference macros would look like:

```
        MACRO
                EMP_ID          = 0,0,%BPVAL/2,0 %,
                EMP_COST_CEN    = 0,%BPVAL/2,%BPVAL/2,0 %,
                EMP_NAME_PTR     = 1,0,%BPVAL,0 %;
```

Based on these macros, we can now write a macro that will take as arguments the initial values and then do the proper packing:

```
        MACRO
                SHIFT(W,P,S,E) = P %,

                EMP_INITIAL(ID,CC,NAME)[] =
                    ID^SHIFT(EMP_ID) OR  ! First
                    CC^SHIFT(EMP_COST_CEN) ,  !

                    NAME_PTR ( NAME^SHIFT(EMP_NAME_PTR)) %;

                                                 ! Second
```

The macro SHIFT simply extracts the position parameter of the field reference macro. The initialization macro, EMP_INITIAL, makes use of this shift value in packing the words. The goal here is to require the user to specify as arguments only the information needed to initialize the table, and not to specify information that is part of its representation.

An example of using these macros to initialize packed data follows:

```
        !+
        !  Employee Field Reference macros
        !-

        MACRO
                EMP_ID  = 0,0,%BPVAL/2,0 %,
                EMP_COST_CEN    = 0,%BPVAL/2,%BPVAL/2,0 %,
                EMP_NAME_PTR    = 1,0,%BPVAL,0 %;

        MACRO

        !+
        !  Macro to create the shift value from the
        !  position parameter of a field reference macro
        !-

                SHIFT(W,P,S,E) = P %,

        !+
        !  Employee table initializing macro
        !  Three values are required
        !-

                EMP_INITIAL(ID,CC,NAME)[] =

                        ID^SHIFT(EMP_ID) OR
                        CC^SHIFT(EMP_COST_CEN) ,   ! First

                        NAME^SHIFT(EMP_NAME_PTR) %;  ! Second

        !+
        !  Employee table definition and initialization
        !-

        OWN
                EMP_TABLE:
                        BLOCKVECTOR[NO_EMPLOYEES, EMP_LINE_SIZE]
                        INITIAL( EMP_INITIAL(
                                345,
                                201,
                                'MARKS PETER',

                                207,
                                345,
                                'NASSI ISAAC'

                                ));
```

What has been illustrated in the previous example is the parameterization of certain values such as field sizes. In transporting this program we can benefit from the localization of certain machine values as in the field reference macros. This code is transportable between BLISS-32 and BLISS-36. To compile this program with the BLISS-16 compiler, we need to change the field reference macros. The packing macros would no longer be needed, though they could be used for consistency purposes. In that case, they would also need to be changed.

As a final example of initializing packed data, we will use another BLOCK structure that is defined in section 12.7.3 of the BLISS-32 Language Guide. Details as to what DCB is and how it accesses data are discussed in the Language Guide. Here, we will only be concerned with initializing this type of structure.

The DCB BLOCK consists of five fields. Four of the fields are packed into one word, their total combined size being 32 bits, and the fifth field which is 32 bits in length occupies another word.

In this case it is possible to transport the DCB initialization very easily between BLISS-32 and BLISS-36. The reason is that the total number of bits required for each word does not exceed the value of %BPVAL for each machine. Hence, in this case at least, we do not have to modify the design of the BLOCK in any way. Typically, however, one would design the structure for each target machine. This is most easily accomplished by placing its definition in a REQUIRE file. We will again make use of the field reference macros as we did in the previous example.

Here is the example showing a way in which it could be initialized. We have extended the structure by making it a BLOCKVECTOR. The example:

```
        !+
        !  DCB size parameters
        !-

        LITERAL
                DCB_NO_BLOCKS = total number of blocks,
                DCB_SIZE = size of a block;

        !+
        !  DCB Field Reference macros
        !-

        MACRO
                DCB_A = 0,0,8,0 %,
                DCB_B = 0,8,3,0 %,
                DCB_C = 0,11,5,0 %,
                DCB_D = 0,16,16,0 %,
                DCB_E = 1,0,32,0 %;
        MACRO

        !+
        ! Macro to create the shift value from the
        ! position parameter of a field reference macro
        !-

                SHIFT(O,P,S,E) = P %,

        !+
        ! DCB initializing macro.
        !  Five values are required.
        !-

                DCB_INITIALIZE(A,B,C,D,E)[] =
                        A^SHIFT(DCB_A) OR
                        B^SHIFT(DCB_B) OR
                        C^SHIFT(DCB_C) OR
                        D^SHIFT(DCB_D) ,

                        E^SHIFT(DCB_E) %;

        !+
        !  DCB Blockvector definition and initialization
        !-

        OWN
                DCB_AREA:
                    BLOCKVECTOR[DCB_NO_BLOCKS, DCB_SIZE]
                    INITIAL(
                            DCB_INITIALIZE (
                            1,2,3,4,
                            5,
                                6,7,8,9,
                                10,
                                ...
```

Note that this structure could be transported to BLISS-16 by making suitable changes to the field reference macros and the packing macro. The only consideration might be whether the last field, DCB_E, did require a full 32 bits.

14.4.5 Structures And Field Selectors

14.4.5.1 Introduction - Two BLISS constructs will be discussed in this section: structures and field selectors. While the use of one does not necessarily imply the use of the other, we will see that for transportability reasons field selector usage will be confined to structure declarations. Hence, these two constructs need to be discussed together.

We will begin with a general discussion of structures, in which it will be shown that a certain machine specific feature of structures can be used in a transportable manner. The best way to illustrate the process of writing transportable structures is to take the reader through the intellectual considerations that contribute to its design, so the development of a transportable structure - FLEX_VECTOR - will be presented. At this point field selectors will be discussed. Finally, a more general structure - GEN_VECTOR - will be developed.

14.4.5.2 Structures -

Structure declarations are sensitive to transportability in that one may specify parameters corresponding to characteristics of particular architectures. Also, in BLISS-32, the reserved words BYTE, WORD, LONG, SIGNED, and UNSIGNED have values of 1, 2, 4, 1 and 0 respectively when used as structure actual parameters.

We can take advantage of the ability to specify architecture-dependent information in developing transportable structure declarations. Later in this section we will develop a structure which will use the UNIT parameter to gain a degree of transportability. The UNIT parameter specifies the number of addressable allocation-units. This number will be used in determining the amount of storage that is to be allocated for each element of the structure.

As mentioned repeatedly in these guidelines, the prime transportability problem is differing machine architectures. Machine word-sizes, for example aren't the same. That is, the number of bits per machine-word differs on all three machines. The machine word is also the maximum size of a BLISS value. There are two other important architectural differences: bits per address and bits per addressable unit.

Bits per address is the maximum size, in bits, of a memory address. Bits per addressable-unit is the size, in bits, of the smallest directly addressable unit in memory.

The values of machine word-size (BLISS value), bits per addressable-unit and bits per address for the three machines have been implemented as predeclared literals, with the names %BPVAL, %BPUNIT and %BPADDR, respectively. A table of their values can be seen in the section entitled: "Literals".

14.4.5.3 FLEX_VECTOR -

We can make use of these values in developing FLEX_VECTOR. First let's state the use to which this structure will be put: We wish to define a structure that will by default allocate and access a vector consisting of only the smallest addressable units. If the default value given in the structure declaration is not used, we want to be able to specify the vector element size in terms of the number of bits. It should be noted that the existing VECTOR mechanism will not do this.

For example, we would like to have a vector of 9-bit elements. The first decision that has to be made is whether or not we want each element to be exactly 9 bits, or at least 9 bits. For this example, we choose the smallest natural unit whose size is greater than or equal to 9 bits. Since there are no 9-bit (in length) addressable units on any of the machines, we have a choice of 8, 16, 32 or 36-bit units.

We can see that 9 bits will fit in the only addressable unit on the 10/20 systems - the word. On the 11 we will need two bytes or a 16-bit word and on the VAX-11 machine we will again need two bytes.

How then do we develop a structure that will do this allocation and will also be transportable and usable on the three systems? Clearly the structure will need some knowledge of the machine architecture. This is where the role of parameterization comes in.

The predeclared literals have all the information we need. In fact we need only one set of values - bits per addressable-unit(%BPUNIT).

This parameter will be one of the allocation formals. Other formals that we will need are the number of elements (N) and the index parmeter (I) for accessing the vector.

We begin by showing the access and allocation formal list for FLEX_VECTOR:

```
        STRUCTURE
                FLEX_VECTOR[ I; N, UNIT = %BPUNIT, EXT = 1 ] =
```

Notice that by setting UNIT equal to %BPUNIT the default (if UNIT is not specified) will be %BPUNIT.

Now we must develop the formula for the structure-size expression. The expression will make use of the allocation formals UNIT and N; and, in addition, the value of the parameter %BPUNIT.

If UNIT were only allowed to assume values of integer multiples of %BPUNIT (i.e. 1*%BPUNIT, 2*%BPUNIT, etc.), we would only need a structure-size expression of the following form:

```
                    [ N * (UNIT) / %BPUNIT ]
```

Dividing the element size (UNIT) by %BPUNIT would give the size of each element in the vector in terms of an integer multiple. This value would then be multiplied by the number of elements to give the total size of the data to be allocated.

We wish, however, for the structure to be more flexible in that we will be able to specify any size element (within certain limits). The structure-size must be slightly more complex:

```
                    [ N * (UNIT + %BPUNIT - 1) / %BPUNIT ]
```

The structure-size expression now computes enough %BPUNIT's to hold the entire vector. The reader should try some values of UNIT for differing %BPUNIT in order to see how this expression evaluates.

This sub-expression:

```
                              (UNIT + %BPUNIT - 1) / %BPUNIT
```

which we will call NO_OF_UNITS is very important in effecting the transportability and flexibility of this particular structure. The key to transporting this structure is the knowledge that it has of a certain machine architectural parameter: bits per addressable-unit. This particular expression makes use of this knowledge, hence, it can adapt to any machine. This sub-expression will be used twice more in the structure-body expression.

The structure-body is an address-expression. This expression will consist of the name of the structure (the base address) plus an offset based on the index I. In addition, a field selector will be needed to access the proper number of bits at the calculated address.

The offset is simply the expression NO_OF_UNITS multiplied by the index I. (Remember that indices start at 0). The size parameter of the field selector is the expression NO_OF_UNITS multiplied by the size of an addressable-unit - %BPUNIT. The structure-body will look like:

```
        (FLEX_VECTOR +
            I * ((UNIT + %BPUNIT - 1) / %BPUNIT))

        <0,((UNIT + %BPUNIT - 1)/%BPUNIT)*%BPUNIT,EXT>;
```

The value of the position parameter in the field-selector is a constant 0 for we are always starting at an addressable boundary.

The following table shows the structure on the three machines for different values of UNIT:

VAX-11	
UNIT = 0	no storage FLEX_VECTOR<0,0,1>
UNIT = 1 to 8	[N * 1] Bytes (FLEX_VECTOR + I)<0,8,1>
UNIT = 9 to 16	[N * 2] Bytes (FLEX_VECTOR + I * 2)<0,16,1>
UNIT = 17 to 32	[N * 4] Bytes (FLEX_VECTOR + I * 4)<0,32,1>
11	
UNIT = 0 to 16	same as VAX-11
10/20	
UNIT = 0	no storage (FLEX_VECTOR)<0,0,1>
UNIT = 1 to 36	[N] Words (FLEX_VECTOR + I)<0,36,1>

From the table above we can see that if the default value for UNIT were set to %BPVAL, this structure would be equivalent to a VECTOR of longwords on VAX-11, and a VECTOR of words on the 10/20 and 11 systems.

Elements in a data segment which has this particular structure attribute are accessed very efficiently because they are always on addressable boundaries. Also, they are always some multiple of an addressable unit in length.

If we wish this structure to access elements exactly the size specified then we need only change the size parameter of the field selector. This expression then becomes:

```
... FLEX_VECTOR<0, UNIT>;
```

This is a less efficient means of accessing data (when UNIT is not a multiple of %BPUNIT) because the compiler needs to generate field selecting instructions in the case of the VAX-11 and 10/20 machines and a series of masks and shifts for the 11.

14.4.5.4 Field Selectors -

In the last structure declaration, it was necessary to make use of a field selector. At this, we will discuss the use of field selectors in a more general context.

The use of field selectors can be non-transportable because they make use of the value of the machine word size. The unrestricted usage of field selectors may cause problems in a program when it is moved to another machine. These problems are best illustrated by the following table of restrictions on position (p) and size (s) for the three machines:

Machine:	10/20	11	VAX-11
	$0 \le p$ $p + s \le 36$ $0 \le s \le 36$	$0 \le p$ $p + s \le 16$ $0 \le s \le 16$ p, s constant	$0 \le s \le 32$

From the table we can see that:

- o The most restrictive is the 11.

- o The moderate restrictions are those of the 10/20.

- o The least restrictive is VAX-11.

If we wished to ensure the transportable use of field selectors, we would have to abide by the set of restrictions imposed in BLISS-16. These, however, are restrictions imposed by the values of p and s. There is also a contextual restriction on the use of field selectors. The following guideline should be followed:

- o Field selectors may only appear in the definition of user-defined structures.

By restricting the domain of field selectors to structures, we are in fact isolating their use.

We will now develop another transportable structure which will be affected by the table of field selector value restrictions.

14.4.5.5 GEN_VECTOR -

You have probably noticed that FLEX_VECTOR does not attempt to pack data. Using the example of 9-bit elements, we can see that there will be some wasting of bits - from 7 bits on the 11 and VAX-11 to 27 on the 10/20 systems.

We can develop a variation of FLEX_VECTOR which will provide a certain degree of packing. For example, in the case of 9-bit elements it would be possible to pack at least four of them into a 10/20 word and three into a VAX-11 longword. Unfortunately, this vector is not maximally transportable, but its design and the identification of its non-transportable aspects should be very helpful.

This structure, which will be named GEN_VECTOR, will pack as many elements as possible into a BLISS value (word) so we will make use of the machine specific literal %BPVAL. But, since allocation is in terms of %BPUNIT, we will need a literal that has as a value the number of allocation units in a BLISS value. This literal has been predeclared for transportability reasons and has the name %UPVAL, and is defined as %BPVAL/%BPUNIT.

Elements will not cross word boundaries. This constraint is in effect because of the restrictions placed on the value of the position parameter of a 10/20 and 11 field selector. For the same reason elements can not be longer than %BPVAL, as given in the table of field selector restrictions above.

As in FLEX_VECTOR, the allocation expression of GEN_VECTOR will need to calculate the number of allocation units needed by the entire vector. This will again be based on the number of elements (N) and the size of each element (S). But because the elements will be packed, the expression will be slightly more complicated.

The first value we need is the number of elements that will fit in a BLISS value. The expression:

```
(%BPVAL/S)
```

will compute this value. Given this, to obtain the number of BLISS values or words needed for the entire vector, we divide this value into N:

```
                    (N/(%BPVAL/S))
```

We now have the total number of words (in units of %BPVAL) needed. However, data is not allocated by words on both of the machines. Multiplying this value by %UPVAL will result in the number of allocation units needed by the vector:

```
                ((N/(%BPVAL/S))*%UPVAL)
```

For clarity's sake and because this expression will be used again we will make it into a macro with N and S as parameters:

```
        MACRO
                WHOLE_VAL(N,S) =

                  ((N/(%BPVAL/S))*%UPVAL)%;
```

The name of the macro suggests that we have calculated the number of whole words needed. If, in fact, N were an integral multiple of the number of elements in a word then this macro would be sufficient for allocation purposes.

Since we can't count on this always happening, we need another expression to calculate the number of allocation units needed for any remaining elements. The number of elements left over is the remainder of the last division in this expression:

```
                    (N/(%BPVAL/S))
```

The MOD function will calculate this value, as follows:

```
                  (N MOD (%BPVAL/S))
```

If we then multiply this value by the size of each element we will have the total number of bits that remain to be allocated:

```
                  (N MOD (%BPVAL/S)) * S
```

This value will always be strictly less than %BPVAL. For the same reasons outlined above we will make this expression into a macro with N and S as parameters:

```
        MACRO
                PART_VAL(N,S) =

                    ((N MOD (%BPAVAL/S)) * S)%;
```

Taking this value, adding a "fudge factor" and then dividing by %BPUNIT will give us the number of allocation units needed for the remaining bits:

```
                (PART_VAL(N,S) + %BPUNIT -1)/%BPUNIT
```

The total number of allocation units has been calculated and the structure allocation expression will look like:

```
                [WHOLE_VAL(N,S) +
                 (PART_VAL(N,S) + %BPUNIT - 1)/%BPUNIT]
```

As it works out, the structure-body expression for GEN_VECTOR will be simple to write because of the expressions that have already been written.

The accessing of an element in GEN_VECTOR requires that we compute an address offset which is then added to the name of the structure. This offset is some number of addressable units based on the value of the index I. We already have an expression which will calculate this number of addressable units. It is the macro WHOLE_VAL. Thus, the first part of the accessing expression will look like:

```
                        GEN_VECTOR + WHOLE_VAL(I,S)
```

Note that the macro was called with the index parameter I.

This expression will result in the structure being aligned on some addressable boundary. But since the element may not begin at this point (that is, the element may be located somewhere within a unit %BPVAL bits in length), one more value is needed. That value is the position parmeter of a field selector. The macro PART_VAL will calculate this value based on the index I:

```
                        <PART_VAL(I,S),S,EXT>
```

The size parameter is the value S. The position parameter will be calculated at run-time, based on the value of the index I. Since I is not constant, we can no longer use this structure in BLISS-16. The position and size parameters of a field selector in BLISS-16 must be compile-time constants. See the table of field selector restrictions above.

This completes the definition of GEN_VECTOR. The entire declaration will look like:

```
STRUCTURE
        GEN_VECTOR[I;N,S,EXT=1] =

            [WHOLE_VAL(N,S) +
             (PART_VAL(N,S) + %BPUNIT - 1)/%BPUNIT]

            (GEN VECTOR + WHOLE_VAL(I,S))

            <PART_VAL(I,S),S,EXT>;
```

The reader should compile this structure and see how it works in BLISS-32 and BLISS-36.

14.4.5.6 Summary -

No claim is made that either of these two structures will solve all the problems associated with transporting vectors. Many such problems will have unique and different solutions. BLOCKS or BLOCKVECTORS have not been discussed, but it is hoped that the reader will get from the examples a feeling for the techniques involved in transporting structures.

There is no easy solution to transporting data structures. One should consider, when developing data structures, the machines that the program or system is targeted for and make full use of the predeclared literals such as %BPUNIT.

This exercise in the development of transportable structures has illustrated two points:

- o parameterization and
- o field selector usage.

By parameterizing certain machine-specific values and by taking full advantage of the powerful STRUCTURE mechanism, we have developed two transportable structures.

The accessing of odd (not addressable) units of data is accomplished by the use of field selectors. The field selector should only be used in structure declarations.

[end chapter 14]

Title: VAX-11 Software Eng. Diagnostic Conventions -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SE15R3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: Diagnostic Coding Conventions

Author: F. Bernaby

Typist: P. Conklin

Reviewer(s): E. Kenney

Abstract: Chapter 15 contains the diagnostic conventional extensions to the rest of this document. It represents an effort to produce diagnostic products in a consistent manner.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	F. Bernaby	Sep-76
Rev 2	skipped to maintain numbers		
Rev 3	Integrated with SE manual	P. Conklin	28-Feb-77

Rev 1 to Rev 3:

1. Just merge file.

2. Update module preface.

[End of SE15R3.RNO]

CHAPTER 15

DIAGNOSTIC CONVENTIONS

28-Feb-77 -- Rev 3

15.1 INTRODUCTION

VAX-11 diagnostics will be written in conformance with the conventions expressed in this manual.

These conventions will be adopted to:

1. achieve clear and meaningful documentation of individual tests.

2. reduce the need for diagnostic users to analyze test code.

3. simplify the program maintenance task.

15.2 DIAGNOSTIC SECTIONS

Each diagnostic will be sub-divided into 15 sections. These sections provide a logical way of partitioning the program.

PROGRAM HEADER	provides the module preface for program
PROGRAM EQUATES	area for macro & symbol definitions
PROGRAM DATA	area for data used by more than one test
PROGRAM TEXT	area for all ASCII messages
PROGRAM ERROR REPORT	area reserved for print module
HARDWARE PTABLE	table of hardware parameters
SOFTWARE PTABLE	table of software parameters
DISPATCH TABLE	table of test addresses for test sequencing
REPORT CODE	print module for statistical reports
INITIALIZE CODE	routine for initializing unit under test(uut)
CLEANUP CODE	routine for cleaning up error states in u
PROGRAM SUBROUTINES	area for routines used by more than 1 test
HARDWARE TEST	actual diagnostic test code
HARDWARE PARAMETERS	code used by supervisor to get hardware ptable entries
SOFTWARE PARAMETERS	code used by supervisor to get software ptable entries

15.2.1 Program Header Section

<EXAMPLE>

```
        .TITLE  SYSEXR - System exerciser
        .IDENT  /2-3/

;
; COPYRIGHT (C) 1977
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
;
; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
; REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;

;++
; FACILITY: diagnostic exerciser
;
; ABSTRACT:
;
;       This program will exercise the VAX-11 system. It generically
;       treats devices as magtape, disk, or terminals.
;       Up to 32 units may be selected for testing.
;
; ENVIRONMENT:  System
;
; AUTHOR: Frank Bernaby, CREATION DATE: 16-Sep-76
;
; MODIFIED BY:
;
;       Joe Hacker, 4-Jul-77: VERSION 2
; 02    -  Added I/O tests for 6250 tape drives.
; 03    -  Brought module preface to standard form.
;--
```

15.2.2 Program Equates(declarations)

<EXAMPLE>

```
;++
; LISTING CONTROL
;--

.NLIST  MC,MD,CND                           ;NO LIST MACRO'S & CONDITIONALS
.LIST   ME                                  ;LIST MACRO EXPANSION

;++
; MACRO LIBRARY CALLS
;--

.MCALL  QIO$S,QIO$C,DPB$,WTEF$C

;++
; INCLUDE FILES: SYSMAC.SML
;--

;++
; EXTERNAL SYMBOLS: DEBUG
;--

.GLOBAL DEBUG                               ;ENTRY POINT OF DEBUGGER


;++
;EQUATED SYMBOLS
;--

;++
; UBA REGISTER DEFINITIONS
;--

UBA_BASE_ADDRESS=177000                     ;UBA BASE ADDRESS
UBA_CSR_OFFSET=0                            ;CONTROL/STATUS REGISTER
UBA_FMR_OFFSET=2                            ;FAILED MAP REGISTER
UBA_IRP_OFFSET=4                            ;MAP REGISTER POINTER
UBA_IRC_OFFSET=6                            ;MAP REGISTER CONTENTS
UBA_SV4_OFFSET=10                           ;REQ SEND VECTOR #4
UBA_SV5_OFFSET=12                           ;REQ SEND VECTOR #5
UBA_SV6_OFFSET=14                           ;REQ SEND VECTOR #6
UBA_SV7_OFFSET=16                           ;REQ SEND VECTOR #7
```

15.2.3 Program Data

<EXAMPLE>

```
;++
; TABLE OF UBA ADDRESS
;
; THIS TABLE IS REFERENCED WHEN ONE OF THE UBA REGISTERS
; MUST BE ADDRESSED. THE UBA REFERENCE IS AN INDIRECT
; REFERENCE THROUGH THIS TABLE. EXAMPLE:
;
;       MOVW    @UBACSR,R0                    ;READ CSR INTO R0
;
;--

TABLE_UBA_ADDRESSES:

UBACSR: .LONG   UBA_BASE_ADDRESS+UBA_CSR_OFFSET
UBAFMR: .LONG   UBA_BASE_ADDRESS+UBA_FMR_OFFSET
UBAIRP: .LONG   UBA_BASE_ADDRESS+UBA_IRP_OFFSET
UBAIRC: .LONG   UBA_BASE_ADDRESS+UBA_IRC_OFFSET
UBASV4: .LONG   UBA_BASE_ADDRESS+UBA_SV4_OFFSET
UBASV5: .LONG   UBA_BASE_ADDRESS+UBA_SV5_OFFSET
UBASV6: .LONG   UBA_BASE_ADDRESS+UBA_SV6_OFFSET
UBASV7: .LONG   UBA_BASE_ADDRESS+UBA_SV7_OFFSET


;++
; DEVICE STATUS BUFFER
;
; THIS AREA IS RESERVED FOR STORING DEVICE STATUS AT
; THE CONCLUSION OF AN I/O OPERATION. THIS STATUS
; IS PROVIDED VIA THE QIO MECHANISM.
;
;--

DEVICE_STATUS:  .BLKL   64                    ;RESERVE 64 LONG WORDS
```

15.2.4 Program Text

<EXAMPLE>

```
;++
; QUESTIONS
;--

QST1_UBA_BASE:            .ASCIZ  %ENTER UBA BASE ADR: %
QST2_UBA_VECTOR:          .ASCIZ  %ENTER UBA VECTOR ADR: %
QST3_UBA_LEVEL:           .ASCIZ  %ENTER UBA BR LEVEL: %
QST4_RECORD_LENGTH:       .ASCIZ  %ENTER RECORD LENGTH: %
QST5_DATA_PATTERN:        .ASCIZ  %ENTER DATA PATTERN: %

                          .
                          .
                          .


;++
; FORMAT STATEMENTS
;--


FMT1_RKCS_DECODE:         .ASCIZ  /%ARKCS: %XW%A : %RW%N/
FMT2_TIMEOUT:             .ASCIZ  /%ATIMEOUT WHILE REFERENCING RK05 REGISTE
XL%N/
FMT3_MACHINE_CHECK:       .ASCII  /%AMACHINE CHECK ABORT: %XW%N%XW%A ITEMS/
                          .ASCIZ  / ON STACK, PC= %XL%A SP= %XL%N/
FMT4_SEEK_ERROR:          .ASCIZ  /%ASEEK BAD ERROR REGISTER: %XW%N/
FMT5_ABORT:               .ASCIZ  /%N%N%APROGRAM ABORTING OPERATION%N/
FMT6_PROG_SUMMARY:        .ASCII  /%NPROGRAM SUMMARY/
                          .ASCII  /%NWORDS TRANSFERRED: %XL/
                          .ASCII  /%NHARD ERRORS: %XL/
                          .ASCIZ  /%SOFT ERRORS: %XL%N/
                             .
                             .
                             .
```

15.2.5 Program Error Report

<EXAMPLE>

```
;++
; PRINT ENTRY POINTS FOR ERROR MESSAGES
;--

MSG1_TIMEOUT:           PRINT   FMT2_TIMEOUT,<R1>         ;PRINT TIMEOUT
                        PRINT   FMT5_ABORT,               ;PRINT ABORT
                        RSB                               ;EXIT


MSG2_MACHINE_CHK:       PRINT   FMT3_MACHINE_CHECK,<R6,R7,(R8),R9>
                        RSB


MSG3_DEV_STATUS:        PRINT   FMT1_RKCS_DECODE,<R3,#BITRKCS>
                        RSB                               ;EXIT
```

15.2.6 Hardware Ptable

<EXAMPLE>

```
;++
; HARDWARE PARAMETER TABLE FOR PROGRAM
;
; THIS TABLE PROVIDES THE REQUIRED HARDWARE PARAMETERS
; FOR TEST EXECUTION. THE ENTRIES ARE OBTAINED FROM EITHER
; THE USER VIA GPHRD COMMANDS OR FROM THE SYSTEM
; CONFIGURATION TABLE.
;
;--


HARD_UBA:

HARD_UBA_BASE:          .LONG   0               ;BASE ADDRESS OF UBA
HARD_UBA_VECTOR:        .LONG   0               ;UBA VECTOR ADDRESS
HARD_UBA_LEVEL:         .LONG   0               ;UBA BR LEVEL

                          .
                          .
                          .
```

15.2.7 Software Ptable

<EXAMPLE>

```
;++
; SOFTWARE PARAMETER TABLE FOR  PROGRAM
;
; THIS TABLE CONTAINS ALL THE REQUIRED SOFTWARE PARAMETERS.
; THESE PARAMETERS ARE OBTAINED VIA GPSFT COMMANDS.
;
;--

SOFT_UBA:

SOFT_RECORD_LENGTH:     .LONG   0               ;RECORD LENGTH
SOFT_DATA_PATTERN:      .LONG   0               ;REQUIRED DATA PATTERN
SOFT_DATA_PATH:         .LONG   0               ;UBA DATA PATH
SOFT_MAP_BASE:          .LONG   0               ;BASE MAP REG TO USE
SOFT_MAP_LENGTH:        .LONG   0               ;# OF MAP REG TO USE

                          .
                          .
                          .
```

15.2.8 Dispatch Table

<EXAMPLE>

```
;++
; PROGRAM DISPATCH TABLE
;
; THIS TABLE IS BUILT BY A SUPERVISOR MACRO
;
;--


TEST_DISPATCH:

                    N                                       ;" N " TEST IN TABLE
                    T1S0                                    ;ADDRESS OF TEST #1
                    T2S0                                    ;ADDRESS OF TEST #2
                    T3S0                                    ;ADDRESS OF TEST #3

                    .
                    .
                    .

                    TNS0                                    ;ADDRESS OF TEST #N
```

15.2.9 Report Code

<EXAMPLE>

```
;++
; STATISTICAL REPORT MODULE
;
; THIS PRINT MODULE PROVIDES REPORTS OF A STATISTICAL NATURE.
; THE FIRST ENTRY IS INVOKED BY THE SUPERVISOR COMMAND 'REPORT'.
; THE REMAINING ENTRIES ARE PROGRAM INVOKED.
;
;--

REP1_PROG_SUMMARY:  PRINT   FMT6_PROG_SUMMARY,<R6,R5,R7>     ;PRINT SUMMAR
Y
                    RSB                                      ;EXIT


REP2_DATA_SUMMARY:  PRINT   FMT7_DATA_SUMMARY,<R3,R4,DATA_TABLE>
                    PRINT   FMT8_DATA_STAT
                    RSB                                      ;EXIT
```

15.2.10 Intialize Code

<EXAMPLE>

```
;++
; FUNCTIONAL DESCRIPTION: INIT
; THIS ROUTINE INITIALIZES THE TEST PROGRAM.
; IT PERFORMS:
; 1. ALLOCATION OF UNIT(S) UNDER TEST
; 2. INITIAL ALLOCATION OF BUFFER SPACE
; 3. INITIAL MAPPING OF MEMORY SPACE
;
; CALLING SEQUENCE: SUPERVISOR INVOKED
;
; INPUT PARAMETERS: PTABLE
;
;--

        BGNINT                                  ;START OF CODE

          .
          .
          .

        ENDINT                                  ;END OF INITIALIZE
```

15.2.11 Cleanup Code

<EXAMPLE>

```
;++
; FUNCTIONAL DESCRIPTION: CLNUP
; THIS ROUTINE PERFORMS THE NECCESSARY CLEANUP BEFORE
; THE TEST PROGRAM EXITS BACK TO SUPERVISOR LEVEL.
; IT PERFORMS:
; 1. DEALLOCATION OF BUFFER SPACE
; 2. RESET OF UNIT UNDER TEST(UUT)
; 3. DEALLOCATION OF UNIT UNDER TEST
;
; CALLING SEQUENCE: JSB CLNUP
;
; INPUT PARAMETERS: PTABLE
;
;--

        BGNCLN                                  ;START OF CLEANUP
          .
          .
          .
        ENDCLN                                  ;END OF CLEANUP
```

15.2.12 Program Subroutines

<EXAMPLE>

```
        .SBTTL  PROGRAM SUBROUTINES


;++
; FUNCTIONAL DESCRIPTION: $RANDOM
; THIS ROUTINE GENERATES A RANDOM NUMBER THAT IS RETURNED
; IN R0. THE SEED FOR THE NUMBER IS PASSED ON THE STACK.
;
; CALLING SEQUENCE: PUSHL    SEED                  ;PUT SEED VALUE ON STCK
                    CALLS    #1,$RAND              ;CALL ROUTINE
;
; INPUT PARAMETERS: SEED
; SEED = BASE VALUE THAT GENERATOR STARTS WITH.
;
;--

$RANDOM:
        .WORD   ^M<R1,R2>                          ;SAVE REG MASK
        MOV     4(AP),R1                           ;FETCH SEED FRM STCK

                .
                .
                .


        MOVL    R1,R0                              ;RETURN VALUE IN R0
$RANDOM_EXIT:

        RET                                        ;RETURN TO CALLER
```

```
;++
; FUNCTIONAL DESCRIPTION: UBA_SETUP
;
; THIS ROUTINE HANDLES THE SETUP OF THE UBA
; TO ALLOW UNIBUS DEVICES TO TRANSFER DATA
; BETWEEN SBI MEMORY AND UNIBUS MEMORY OR
; UNIBUS DEVICES
;
; CALLING SEQUENCE: CALLG       #UBA_LIST,$UBA_SETUP
;
; INPUT PARAMETERS:      UBA_LIST
;
; THIS LIST IS A TABLE LIKE:
;
;       UBA_LIST:               5                       ;NUMBER OF ARGUMENTS
;       UBA_BUS_ADR:            .LONG   0               ;BUS ADR AT DEVICE
;       UBA_LENGTH:             .LONG   0               ;RECORD LENGTH
;       UBA_MAP_BASE:           .LONG   0               ;STARTING MAP REG
;       UBA_DAT_PATH:           .LONG   0               ;UBA DATA PATH
;       UBA_SBI_PHYSICAL:       .LONG   0               ;STARTING PHYSICAL ADR
;
;--


$UBA_SETUP:

        .WORD   ^M<R1,R2,R3,R4>         ;SAVE R1-R4
        MOVL    4(AP),R1                ;GET ADDR OF ARGUMENT LIST

        .
        .
        .

$UBA_SETUP_EXIT:

        RET                             ;EXIT
```

15.2.13 Hardware Test

The actual harware test will go within this section of the program. All diagnostics that run with the diagnostic supervisor will, when neccessary, make supervisor 'calls' to provide a function rather than code that function into the program.

If a routine is used by more than one test, that routine will be placed in the program subroutine section. Linkage to that routine will be via 'CALLS' or 'CALLG' instructions. If these routines must pass data back to the test, the test will specify where this data will go by supplying the needed argument(s).

This section is sub-divided by tests and subtests. The test subdivision provides for blocking the diagnostic of into major logic areas. While, the subtest provides a way of further subdividing each test into smalller logic areas.

Therefore the basic organization will look like this.

```
        BGNTST

                BGNSUB

                <TEST CODE FOR T1S1>

                ENDSUB
                BGNSUB

                <TEST CODE FOR T1S2>

                ENDSUB

        ENDTST
        BGNTST

                BGNSUB

                <CODE FOR T2S1>

                ENDSUB

        ENDTST
```

Each test and subtest must have a specific level of documentation.

Each test must specify a complete test description and any assumptions that are assumed by this test. Assumptions implies what logic is assumed to have been successful tested when this test starts.

Each subtest must have the test description and assumptions. In addition, the subtest must have a complete description of how the subtest works, what errors the subtest will detect, and what the debug procedure is for the subtest failure.

<EXAMPLE>

```
        BGNTST

;++
;
; TEST DESCRIPTION:
;
; THIS TEST CHECKS THE MAP REGISTERS IN THE UBA. IT PERFORMS THIS TEST
; BY CHECKING THAT ALL REGISTERS HOLD ZEROS AND ONES. THEN THE TEST
; WILL FLOAT A ONE THROUGH ALL REGISTERS. FINALLY, THE TEST WILL FLOAT
; A ZERO THROUGH ALL REGISTERS
;
; ASSUMPTIONS:
;
; TEST1-TEST2
; THIS TEST ASSUMES THAT THE DATA PATH FROM THE CPU TO THE UBA
; HAS BEEN CHECKED AND THAT REGISTER ADDRESSING WORKS CORRECTLY.
;
;--


T3S0:
```

```
BGNSUB

;++
;
; TEST DESCRIPTION:
;
; THIS SUBTEST CHECKS THAT UBM000-UBM496 WILL
; HOLD AN ALL ZEROS DATA PATTERN AND AN ALL ONES DATA
; PATTERN.
;
; ASSUMPTIONS:
;
; TEST1-TEST2
;
; TEST STEPS:
;
; 1. INIT MAP REGISTER INDEX TO ZERO(R3)
; 2. CLEAR SELECTED MAP REGISTER-MP(R3)
; 3. IF MP(R3) .EQU 0 THEN CONTINUE ELSE REPORT ERROR
; 4. COMPLEMENT SELECTED REGISTER-MP(R3)
; 5. IF MP(R3) .EQU -1 THEN CONTINUE ELSE REPORT ERROR
; 6. SELECT NEXT REGISTER(UPDATE R3)
; 7. IF R3 .GTR 496 THEN EXIT ELSE GOTO STEP 2
;
; ERRORS:
;
; 1. TIMEOUT- UBA FAILED TO RESPOND
; 2. ZEROS DATA FAILURE
; 3. ONES DATA FAILURE
;
; DEBUG:
;
; ERROR #1-
; THIS ERROR COULD MEAN POWER FAILURE. CHECK SUPPLIES
;
; ERROR #2-
; CHECK BIT(S) THAT FAILED FOR STUCK AT ONE STATE
;
; ERROR #3-
; CHECK BIT(S) THAT FAILED FOR STUCK AT ZERO STATE
;
;--


T3S1:

<TEST CODE>

T3S1X:
ENDSUB
```

15.2.14 Hardware Parameter Code

<EXAMPLE>

```
;++
; THE HARDWARE PARAMETER TABLE IS BUILT FROM THE INSTRUCTIONS
; IN THIS SECTION. THESE INSTRUCTIONS GET EXECUTED IF THE USER
; STARTS THE PROGRAM WITHOUT SPECIFYING A CONFIGURATION TABLE.
; THE SUPERVISOR WILL RECOGNIZE THIS AN DISPATCH TO THIS SECTION.
; THE INPUT TO THESE REQUEST CAN COME FROM EITHER THE USER
; OR A SCRIPT FILE.
;
; INPUT IS ELICITED BY GPHRD. THIS COMMAND HAS THE FOLLOWING
; FORMAT:
; GPHRD (TABLE OFFSET,FORMAT STATEMENT,RADIX,BYTE OFFSET,LOWER LIMIT,
;        UPPER LIMIT)
;
;--


        BGNHRD                                    ;BEGINNING OF HARDWARE CODE

HPM1:   GPRMD    (BASADR,QST1,O,1,177000,177170)  ;GET UBA BASE ADR
HPM2:   GPRMD    (VCTADR,QST2,O,1,100,400)        ;GET UBA VECTOR ADR
            .
            .
            .
        ENDHRD                                    ;RETURN TO SUPERVISOR
```

15.2.15 Software Parameter Code

<EXAMPLE>

```
;++
; THE SOFTWARE PARAMETER TABLE IS BUILT FROM THIS CODE IF THE
; DIAGNOSTIC SUPERVISOR IS DIRECTED TO ACCEPT SOFTWARE
; PARAMETERS FROM EITHER A SCRIPT FILE OR THE USER.
; GPSFT COMMANDS ARE USED TO BUILD THE TABLE. THE FOMRAT
; OF THE ARGUMENTS IS THE SAME AS FOR GPHRD(SEE 8.2.14).
;
;--


        BGNSFT                                    ;FETCH SOFTWARE PARAMS


SPM1:   GPRMD    (RCDLEN,QST4,0,1,20,2000)        ;GET RECORD LENGTH
SPM2:   GPRMD    (DATPTN,QST5,O,1,1,17)           ;GET DATA PATTERN
            .
            .
            .
        ENDSFT                                    ;RETURN TO SUPERVISOR
```

15.3 SYMBOL CONVENTIONS

The following symbol conventions should be used for all VAX-11 diagnostics:

TnSm	specifies test n subtest m
FMTn	specifies format statement n
ASCn	specifies ASCII string n
MSGn	specifies error message n
REPn	specifies statistical report n
QSTn	specifies question n
ISRn	specifies interrupt service routine n
SPMn	software parameter n
HPMn	hardware parameter n

The symbol construction should be as follows:

<prefix>_<descriptive name>_<optional modifier>

15.4 MACRO EXPANSION CONVENTIONS

Macros will be expanded or not expanded based on the following rules. If a macro generates inline test code it will be expanded but not it's call. If a macro makes a call to a subroutine the macro call is shown but not it's expansion. Both the call and expansion can be displayed it the program is assembled with a debug switch set.

[End of Chapter 15]

Title: VAX-11 Assembler Software Engineering Sample -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SEAR3.RNO

PDM #: not used

Date: 28-Feb-77

Superseded Specs: none

Author: P. Conklin

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, S. Poulsen, D. Tolman

Abstract: Appendix A contains a copy of a sample module written in assembly language.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	28-Feb-77

Rev 2 to Rev 3:

1. Added example.

[End of SEAR3.RNO]

APPENDIX A

ASSEMBLER SAMPLE

28-Feb-77 -- Rev 3

The listing on the next page shows a routine from the procedure library. There is no suggestion that this routine actually works, only that it follows the conventions set forth in this document. In fact, its "facility" does not even exist. Note that it consists of two externally callable routines and a number of internal routines.

```
        .TITLE  CHF$SIGNAL - Condition Handling Facility SIGNAL and STOI
        .IDENT  /1-3/

;
; COPYRIGHT (C) 1977
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
;
; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
; REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;

;++
; FACILITY:     Condition Handling
;
; ABSTRACT:
;
;       The Condition Handling Facility supports the exception
;       handling mechanisms needed by each of the common languages.
;       It provides the programmer with some control over fixup,
;       reporting, and flow of control on errors.  It provides
;       subsystem and application writers with the ability to
;       override system messages in order to give a more suitable
;       application oriented interface.
;
;       To understand CHF more fully, refer to its functional
;       specification and to the STARLET exception routine (EXCEPTION).
;
;
; ENVIRONMENT:  Any access mode--normally user mode
;
; AUTHOR: Peter F. Conklin, CREATION DATE: 12-Nov-76
;
; MODIFIED BY:
;
;       Peter F. Conklin, 5-Jan-77: VERSION 01
; 01    -  Original, based on CHF Rev 4 spec
; 02    -  (CVC) Updated to Rev 2 coding standards
; 03    -  Correct code in internal handler.
;--
```

```
        .SBTTL  DECLARATIONS
;
; INCLUDE FILES:
;
        $PSLDEF                                 ;PSL definitions
        $SSDEF                                  ;System Status code definitions

;
; MACROS:
;
;       NONE
;
; EQUATED SYMBOLS:
;

CANT_MSG_CTRL_L=40                              ;length control string for CHF$STOP
CANT_MSG_BUF_L=40                               ;length insert message for CHF$STOP


CHF$_=^X2222@16                                 ;***Temp*** CHF facility code

CHF$_CANT_CONT==CHF$_+4                         ;Can't continue from CHF$STOP
CHF$_NO_HANDLER==CHF$_+8                        ;No handler found

SRM$L_HANDLER=0                                 ;Call frame handler
SRM$W_SAVE_PSW=4                                ;Call frame PSW
SRM$W_SAVE_MASK=6                               ;Call frame save mask
SRM$L_SAVE_AP=8                                 ;Call frame save AP
SRM$L_SAVE_FP=12                                ;Call frame backward link
SRM$L_SAVE_PC=16                                ;Call frame save PC

;
; OWN STORAGE:
;
;       NONE
```

```
          .SBTTL  CHF$STOP -  Stop execution via signalling
;++
; FUNCTIONAL DESCRIPTION:
;
;       This procedure is called whenever it is impossible
;       to continue execution and no recovery is possible.
;       It signals the exception. If the handler(s) return
;       with a continue code, a message "Can't continue"
;       is issued and the image is exitted. This procedure
;       is guaranteed to never return.
;
;
; CALLING SEQUENCE:
;
;       CALL CHF$STOP (condition_value.rlc.v, [{parameters.rz.v}])
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       The process is EXITted if a handler specifies continue.
;
;--

          $FORMAL          <-
CONDITION_VALUE-        ;CONDITION_VALUE.rlc.v is the condition
        >               ;other arguments are parameters
```

```
   .ENTRY  CHF$STOP,^M<R2>          ;Stop
BSBB    SIGNAL                      ;go do the signaling
MOVAL   -CANT_MSG_CTRL_L(SP),SP     ;allocate room for control string
PUSHAB  (SP)                        ;set pointer to it
PUSHL   #CANT_MSG_CTRL_L            ;make into string descriptor
MOVL    SP,R2                       ;save a copy of descriptor
$GETERR_S CONDITION_VALUE(AP),(R2),(R2)
                                    ;get error string
MOVAL   -CANT_MSG_BUF_L(SP),SP      ;allocate room for string
PUSHAB  (SP)                        ;get pointer to it
PUSHL   #CANT_MSG_BUF_L             ;make into string descriptor
MOVL    SP,R0                       ;get pointer to it
PUSHL   R0                          ;set as arg for later
$FAOL_S (R2),(R0),(R0),CONDITION_VALUE(AP)
                                    ;format error string
PUSHL   #CHF$_CANT_CONT             ;set "can't continue" code
CALLS   #2,LIB$OUT_MESSAGE          ;issue message, with the
                                    ; original SIGNAL's message
                                    ; as the insert
BRW     SIG_EXIT                    ;stop with original exception
                                    ; as the code
```

```
        .SBTTL  CHF$SIGNAL -  Signal Exceptional Condition
;++
; FUNCTIONAL DESCRIPTION:
;
;       This procedure is called whenever it is necessary
;       to indicate an exceptional condition and the procedure
;       can not return a status code. If a handler returns
;       with a continue code, CHF$SIGNAL returns with
;       all registers including R0 and R1 preserved.  Thus,
;       CHF$SIGNAL can also be used to plant performance and
;       debugging traps in any code.  If no handler is found,
;       or all resignal, a catch-all handler is CALLed.
;
;
; CALLING SEQUENCE:
;
;       CALL CHF$SIGNAL (condition_value.rlc.v, [{parameters.rz.v}])
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       If a handler unwinds, then control will not return.
;       A handler could also modify R0/R1 and change the
;       flow of control.  If neither is done, then all
;       registers and condition codes are preserved.
;
;--


                .ENTRY  CHF$SIGNAL,0            ;Signal
        BSBB    SIGNAL                  ;go do the signaling
        RET                             ;return to caller
        .SBTTL  SIGNAL -  Internal Routine to Signal Exceptions
;++
; FUNCTIONAL DESCRIPTION:
;
;       This routine is used by CHF$STOP and CHF$SIGNAL to do
;       the actual exception signaling. It sets up the handler
;       argument list. It then checks both exception vectors for
;       a handler. It then searches backward up the stack, frame
;       by frame looking for a handler. Each handler found is
;       called. If the handler returns failure (resignal), the
;       search continues. If no handler is found or if all handlers
;       resignal a catch-all handler is called. The catch-all
;       issues the standard message for the condition and then
;       returns success if condition-value<0> is set. If a
;       handler returns success (continue) the routine returns
;       to CHF$STOP or CHF$SIGNAL with R0/R1 intact.
;
;       During the stack search, if another signal is found to
;       be still active, the frames up to and including the
;       establisher of the handler are skipped. Refer to the
;       section Multiply Active Signals in the functional
;       specification. An active signal is defined as a routine
;       which is called from the system vector SYS$CALL HANDLR.
;
;       If a memory access violation is found during the stack
;       search, it is assumed that the stack is finished and
;       the routine calls the catch-all handler.
;
;
; CALLING SEQUENCE:
;
;       JSB
;
; INPUT PARAMETERS:
;
;       AP points to the arg list
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
```

```
;
; SIDE EFFECTS:
;
;       If a handler unwinds, then control will not return.
;       A handler could also modify R0/R1 and change the flow
;       of control. If neither is done, then all registers
;       are preserved.
;
;--



SIGNAL:
        PUSHR   #^M<R0,R1>                  ;save R0/R1 in mechanism vector
        MOVAB   W^SIGNAL_HANDLER,SRM$L_HANDLER(FP) ;establish a handler
                                            ; to catch access violations
        MNEGL   #3,-(SP)                    ;initial depth is -3
        PUSHL   FP                          ;vector frame = current
        PUSHL   #4                          ;mechanism has 4 elements
        PUSHAL  (SP)                        ;second arg is mechanism vector
        PUSHAL  (AP)                        ;first arg is signal vector
        PUSHL   #2                          ;two arguments to handler

;
;  At this point the stack is all set for a call to any handler:
;
;       00(SP) = 2
;       04(SP) = signal vector address
;       08(SP) = mechanism vector address
;       12(SP) = mechanism vector length (4)
;       16(SP) = mechanism vector frame (FP)
;       20(SP) = mechanism vector depth (-3)
;       24(SP) = mechanism vector R0
;       28(SP) = mechanism vector R1
;       32(SP) = RSB return to CHF$STOP or CHF$SIGNAL
;       36(SP)++ RET frame to invoker
;

;
; loop here looking for a handler to call
;

10$:    INCL    20(SP)                      ;move to next depth
        BGEQ    20$                         ;branch if searching stack
        MOVPSL  R0                          ;get current PSL
        EXTZV   #PSL$V_CURMOD,#PSL$S_CURMOD,R0,R0
                                            ;get current mode
        MOVQ    @#CTL$AQ_EXCVEC[R0],R0      ;get both exception vectors
        CMPB    #-1,20(SP)                  ;see which vector this time
        BEQL    40$                         ;branch if secondary
        MOVL    R0,R1                       ;if primary, move to R1
        BRB     40$                         ; and branch
```

```
;
; here if searching stack
;

20$:    BLBS    22(SP),SIGNAL_CATCH     ;if loop too long, give up
        MOVL    16(SP),R0               ;get last frame examined
        MOVL    SRM$L_SAVE_FP(R0),16(SP) ;get previous frame
        BEQL    SIGNAL_CATCH            ;branch if no more stack

;
; Here with R0 containing a frame whose predecessor might be
; CHF or EXCEPTION calling to a handler. If so, the return
; PC would be SYS$CALL_HANDL+4 because both JSB to that
; vector to call handlers. If so, we have the situation of
; multiply active signals and need to bypass frames until this
; handler's establisher is skipped. The depth parameter is
; not incremented because a handler and its establisher are
; considered part of the same entity.
;

        CMPL    SRM$L_SAVE_PC(R0),#SYS$CALL_HANDL+4
                                        ;see if multiply active
        BNEQU   30$                     ;branch if not
        BSBB    OLD_SP                  ;adjust R0 to what SP
                                        ; contained before the call
        MOVL    12(R0),R0               ;get mechanism vector
        MOVL    4(R0),16(SP)            ;get establisher's frame
                                        ; as last frame
        BRB     20$                     ;search again

30$:    MOVL    @16(SP),R1              ;get handler if any
40$:    TSTL    R1                      ;see if handler
        BEQL    10$                     ;if no handler, loop
        JSB     @#SYS$CALL_HANDL        ;CALL handler via "vector"
        BLBC    R0,10$                  ;if resignal, loop
        MOVAL   -12(FP),SP              ;clean up stack
        POPR    #^M<R0,R1>              ;restore R0/R1
        RSB                             ;return to CHF$STOP or CHF$SIGNAL

;
; Here when no handler is found, or if all handlers resignal.
; This is either done when the stack saved FP is 0, meaning end
; of the stack, or when an access violation occurs, indicating
; that the stack is bad. The catch-all handler is called and
; then a no-handler message is issued.
;

SIGNAL_CATCH:
        MOVAB   B^SIG_CATCH_ALL,R1      ;set address of handler
        JSB     @#SYS$CALL_HANDL        ;CALL handler via "vector"
        PUSHL   #CHF$_NO_HANDLER        ;get "no handler" code
        CALLS   #1,LIB$OUT_MESSAGE      ;output message
        BRB     SIG_EXIT                ;go exit with condition
                                        ; value as result
        .SBTTL  SIG_CATCH_ALL -  Internal Catch-all Handler
;++
; FUNCTIONAL DESCRIPTION:
;
;       This handler is used in SIGNAL to catch
;       signals when no handler is found or all resignal.
;
; CALLING SEQUENCE:
;
;       handled = SIG_CATCH_ALL (condition.rl.ra, mechanism.rl.ra)
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       If condition_value<0> is clear, $EXIT is done.
;
;--


SIG_CATCH_ALL:
        .WORD   0                       ;No registers
        MOVAL   @4(AP),AP               ;get condition args
        CALLG   (AP),LIB$OUT_MESSAGE    ;issue standard message
        BLBC    CONDITION_VALUE(AP),SIG_EXIT
                                        ;if failure, go exit
        RET                             ; otherwise, return

;
; Here to give up and exit to the system. The condition value
; argument is given as the exit status.
;

SIG_EXIT:
        $EXIT_S CONDITION_VALUE(AP)     ;exit with condition
                                        ; value as the result
        .SBTTL  OLD_SP -  Internal Routine to Calculate Old SP
;++
; FUNCTIONAL DESCRIPTION:
;
;       This routine is called to calculate what SP was before
;       a particular CALL that resulted in a specific stack
;       frame.
;
;
; CALLING SEQUENCE:
;
;       JSB
;
; INPUT PARAMETERS:
;
;       R0 = address of stack frame in question
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       R0 = value of SP before CALL in question
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       NONE
;
; SIDE EFFECTS:
;
;       R1 is clobbered
;
;--


OLD_SP:
        EXTZV   #14,#2,SRM$W_SAVE_MASK(R0),-(SP)
                                        ;get stack offset
        EXTZV   #0,#12,SRM$W_SAVE_MASK(R0),R1
                                        ;get register mask
        ADDL2   #20,R0                  ;standard frame
        ADDL2   (SP)+,R0                ;SP correction
10$:    BLBC    R1,20$                  ;if register bit set,
        ADDL2   #4,R0                   ; count the register
20$:    ASHL    #-1,R1,R1               ;discard bit
        BNEQU   10$                     ;loop until all done

        RSB                             ;return
        .SBTTL  SIGNAL_HANDLER -  Internal Routine to Handle Access Violation
;++
; FUNCTIONAL DESCRIPTION:
;
;       This handler is used in SIGNAL to catch
;       access violations during the stack search.
;       If it gets an access violation exception from
;       this procedure it terminates the search.
;
;
; CALLING SEQUENCE:
;
;       handled = SIGNAL_HANDLER (condition, mechanism)
;
; INPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT INPUTS:
;
;       NONE
;
; OUTPUT PARAMETERS:
;
;       NONE
;
; IMPLICIT OUTPUTS:
;
;       NONE
;
; COMPLETION CODES:
;
;       0 if not handled
;       success if unwound
;
; SIDE EFFECTS:
;
;       The stack is unwound and SIGNAL_CATCH is branched to.
:
;--
```

```
   SIGNAL HANDLER:
        .WORD   0                       ;No registers
        MOVQ    4(AP),R0                ;get both arguments
        TSTL    8(R1)                   ;verify "this" establisher
        BNEQU   10$                     ;branch if not
        CMPL    4(R0),#SS$_ACCVIO       ;see if memory access violation
        BNEQU   10$                     ;branch if not

   ;
; here if access violation in signal procedure
;

           MOVL    CHF$L_SIG_ARGS(R0),R1   ;get number of signal args      ;M03
        MOVAL   SIGNAL_CATCH,-4(R0)[R1] ;change PC of exception         ;M03
        MOVL    $SS$_CONTINUE,R0        ;resume                         ;M03
        RET                             ; execution                     ;M03
  10$:  CLRL    R0                      ;not handled function value
        RET                             ;return to unwind


                   .END
```

[End of Appendix A]

Title: VAX-11 BLISS Software Engineering Sample -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SEBR3.RNO

PDM #: not used

Date: 27-Feb-77

Superseded Specs: none

Author: P. Conklin

Typist: P. Conklin

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, D. Tolman

Abstract: Appendix B contains a copy of a sample module written in BLISS.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	M. Spier	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	P. Conklin	27-Feb-77

Rev 2 to Rev 3:

1. Added example.

[End of SEBR3.RNO]

APPENDIX B

BLISS SAMPLE

27-Feb-77 -- Rev 3

The listing on the next page shows a routine from the procedure library. There is no suggestion that this routine actually works, only that it follows the conventions set forth in this document.

```
MODULE LIB$OUT_MESSAGE ( !Library routine to output a system message
                IDENT='1-4'
                ) =
BEGIN

!
! COPYRIGHT (C) 1977
! DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
!
! THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
! COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
! ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
! MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
! EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
! TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
! REMAIN IN DEC.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
! CORPORATION.
!
! DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
!

!++
! FACILITY:     Procedure Library
!
! ABSTRACT:
!
!       This routine takes a system message (status) code, gets
!       it from the system message file and formats it with FAO.
!       It then outputs the message to OUTPUT.
!
! ENVIRONMENT:  Any access mode--normally user mode
!
! AUTHOR: Peter F. Conklin, CREATION DATE: 16 Dec 76
!
! MODIFIED BY:
!
!       Peter F. Conklin, 29-Dec-76: VERSION 01
! 01    -  Original, using QIO to TT: only.
! 02    -  Update to standard module format
! 03    -  Change to use GETERR_FRST and GETERR_NEXT
!          and to use PUT_SYSOUT
! 04    -  (CVC) Correct sense of multi-line loop.
!--
```

```
!
! TABLE OF CONTENTS:
!

FORWARD ROUTINE
        LIB$OUT_MESSAGE:NOVALUE; !output message

!
! INCLUDE FILES:
!
!       NONE
!
! MACROS:
!
!       NONE
!
! EQUATED SYMBOLS:
!

LITERAL
        MSG_CTRL_L=132,         !length of control string
        MSG_BUF_L=132;          !length of message

!
! OWN STORAGE:
!
!       NONE
!
! EXTERNAL REFERENCES:
!

EXTERNAL ROUTINE
        LIB$GETERR_FRST:NOVALUE, !get start of message
        LIB$GETERR_NEXT,        !get more of message                    !A03
        SYS$FAOL:NOVALUE,       !format message
        LIB$PUT_SYSOUT:NOVALUE; !put message to SYSOUT:                 !A03
```

```
GLOBAL ROUTINE LIB$OUT_MESSAGE (        !Output system message
        MESSAGE_CODE,   !standard completion code
        LIST)           !substitutable params
        :NOVALUE =

!++
! FUNCTIONAL DESCRIPTION:
!
!       This routine takes a system message (status) code, gets
!       each line of the message from the system message file
!       via the library routines GETERR_FRST and GETERR_NEXT,
!       formats it with FAO, and outputs it via the library
!       routine PUT_SYSOUT.
!
! FORMAL PARAMETERS:
!
!       MESSAGE_CODE.rlc.v      <31:16> facility code
!                               <15:3>  message indicator
!                               <2:0>   severity indicator:
!                                       0 = warning
!                                       1 = success
!                                       2 = error
!                                       4 = severe error
!
!       [{LIST.rz.v}]  remaining parameters are used in call to FAO
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       One or more records are output on device OUTPUT:
!
!--

    BEGIN

    LOCAL
        CONTROL,                    !message line control code
        MSG_CTRL:VECTOR[CH$ALLOCATION(MSG_CTRL_L)], !control string
        MSG_BUF:VECTOR[CH$ALLOCATION(MSG_BUF_L)], !text string
        MSG_CTRL_D:VECTOR[2],     !control string descriptor
        MSG_BUF_D:VECTOR[2];      !text string descriptor
    !
    ! Initialize string descriptors
    !

    MSG_CTRL_D[0] = MSG_CTRL_L;
    MSG_CTRL_D[1] = MSG_CTRL;
    MSG_BUF_D[1] = MSG_BUF;

    !
    ! Get the message control string for the first line
    !

    LIB$GETERR_FRST (.MESSAGE_CODE, MSG_CTRL_D, MSG_CTRL_D, CONTROL);     !A03

    !+
    ! Loop, processing each line and getting the next
    ! The loop ends when GETERR_NEXT returns false
    !-

    DO                                                                   !A03
        BEGIN                                                            !A03

        !+
        ! If the control code is ' ' or 'T', then
        ! format message for output with FAO and then output it.
        ! Note that GETERR returns the control code as uppercase only.
        !-

        IF .CONTROL<0,8> EQLU ' ' OR .CONTROL<0,8> EQLU 'T'              !A03
        THEN
            BEGIN
            MSG_BUF_D[0] = MSG_BUF_L;
            SYS$FAOL (MSG_CTRL_D, MSG_BUF_D, MSG_BUF_D, MESSAGE_CODE);
            LIB$PUT_SYSOUT (MSG_BUF_D);                                  !A03
            END;

        !
        ! Reset control text length in descriptor and get next line
        !

        MSG_CTRL_D[0] = MSG_CTRL_L;                                      !A03
        END                                                              !A03
    WHILE LIB$GETERR_NEXT (MSG_CTRL_D, MSG_CTRL_D, CONTROL);             !M04
                                                                         !A03

    END;                        !End of LIB$OUT_MESSAGE
```

```
END                                     !End of module
ELUDOM
```

[End of Appendix B]

Title: COMMON BLISS Software Engineering Sample -- Rev 3

Specification Status: draft

Architectural Status: under ECO control

File: SECR3.RNO

PDM #: not used

Date: 27-Feb-77

Superseded Specs: none

Author: R. Murray

Typist: R. Murray

Reviewer(s): R. Brender, D. Cutler, R. Gourd, T. Hastings, I. Nassi, D. Tolman

Abstract: Appendix C contains a copy of a sample module written in BLISS.

Revision History:

Rev #	Description	Author	Revised Date
Rev 1	Original	P. Belmont	14-Apr-76
Rev 2	Revised from Review	P. Marks	21-Jun-76
Rev 3	After 6 months experience	R. Murray I. Nassi	27-Feb-77

Rev 2 to Rev 3:

1. Added example.

[End of SECR3.RNO]

APPENDIX C

COMMON BLISS SAMPLE

27-Feb-77 -- Rev 3

The following is a runninq BLISS program that illustrates many of the conventions discussed in this manual. It relies on a small number of external routines for console I/O. These are:

```
TTY_GET_CHAR
TTY_PUT_CHAR
TTY_PUT_CRLF
TTY_PUT_INTEGER
TTY_PUT_ASCIZ
TTY_PUT_MSG
```

```
MODULE LIB$CALC (       ! INTEGER ARITHMETIC EXPRESSION EVALUATOR
                IDENT = '03',
                MAIN = MAINLOOP
                ) =
BEGIN

! COPYRIGHT 1976, DIGITAL EQUIPMENT CORP., MAYNARD, MA 01754
!
! THIS SOFTWARE IS FURNISHED TO THE PURCHASER UNDER  A  LICENSE
! FOR  USE  ON A SINGLE COMPUTER SYSTEM AND CAN BE COPIED (WITH
! INCLUSION OF DIGITAL'S COPYRIGHT NOTICE) ONLY FOR USE IN SUCH
! SYSTEM,  EXCEPT  AS  MAY  OTHERWISE BE PROVIDED IN WRITING BY
! DIGITAL.
!
! THE INFORMATION IN THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT
! NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL
! EQUIPMENT CORPORATION.  DIGITAL ASSUMES NO RESPONSIBILITY FOR
! ANY ERRORS THAT MAY APPEAR IN THIS DOCUMENT.
!
! DIGITAL EQUIPMENT CORPORATION ASSUMES NO  RESPONSIBILITY  FOR
! THE  USE OR RELIABLILTY OF ITS SOFTWARE ON EQUIPMENT WHICH IS
! NOT SUPPLIED BY DIGITAL EQUIPMENT CORPORATION.
!

!++
!
! FACILITY: GENERAL LIBRARY
!
! FUNCTIONAL DESCRIPTION:
!       THIS PROGRAM PARSES AND EVALUATES ARITHMETIC EXPRESSIONS,
!       KEEPS 26 VALUES AROUND, AND GENERALLY ACTS LIKE AN "AID"
!       WITH DECIMAL INTEGERS ONLY.
!
! ENVIRONMENT: USER MODE WITH EXTERNAL ROUTINES
!
!
!
! AUTHOR: P. BELMONT    CREATION DATE: 01-JAN-76
!
! MODIFIED BY:
!       PETER C. MARKS, 10-MAY-76
! 01    -  CONFORMATION TO S. E. MANUAL STANDARDS
!
!       RICHARD M. MURRAY, 21-FEB-77
! 01    -  CONFORM TO REVISED STANDARD
!
!       ISAAC R. NASSI, 30-APR-77
! 01    -  BUG FIXES, TRANSPORTABILITY CHANGES
!
!
!--
```

```
! EXTENDED FUCTIONAL DESCRIPTION:
!
!       SYNTAX:
!
!       LEXICAL LEVEL: ALL CHARACTERS WITH ASCII VALUE LEQ #040
!       ARE IGNORED.  THUS, BLANKS AND TABS AND <CR> AND <NL>
!       ARE IGNORED (AND MANY OTHERS).
!       UPPER AND LOWER CASE ALPHABETIC CHARACTERS ARE IDENTIFIED.
!
!       SINCE WE READ AHEAD ONE CHARACTER, THE USER MUST
!       TYPE SOMETHING AFTER THE LAST CHARACTER TO GET THE JOB DONE.
!       AFTER PROCESSING, THE REMAINDER OF THE INPUT IS ERASED.
!
!       THE UNARY MINUS ( <T1> ) MAY NOT IMMEDIATELY FOLLOW
!       ANY OPERATOR EXCEPT "(".  THUS -1+1;     (-1+1);
!       (-1+(-2));    ARE ALL CORRECT BUT  -1+-2;     IS NOT.
!
!       <FULL> ->        <EXPR> ;
!       <EXPR> ->        <ALPHA>=<EXPR> ! <T5>
!       <T5>   ->        <T5> + <T4> ! <T4>
!       <T4>   ->        <T4> - <T3> ! <T3>
!       <T3>   ->        <T3> / <T2> ! <T2>
!       <T2>   ->        <T2> * <T1> ! <T1>
!       <T1>   ->        - <T0> ! <T0>   (SEE COMMENT ABOVE ON USE
!                                                   UNARY MINUS.)
!       <T0>   ->        ( <EXPR> ) ! <ALPHA> ! <DECIMAL>
!       <ALPHA> ->       A ! B ! C ! ... ! Z
!       <DECIMAL> ->     <DECIMAL><DIGIT> ! <DIGIT>
!       <DIGIT> ->       0 ! 1 ! 2 ! ... ! 7 ! 8 ! 9
!
!       SEMANTICS:
!
!       THERE ARE 26 VARIABLES WITH <ALPHA> NAMES.  THEY
!       ARE INITIALLY ZERO.
!
!       ASSIGNMENT (THE "=" OPERATOR) IS ALLOWED ONLY TO
!       A VARIABLE AND HAS THE EFFECT OF REPLACING THE VALUE
!       OF THE VARIABLE WITH THE EVALUATED VALUE OF THE  <T5>.
!       THE VALUE OF AN ASSIGNMENT OPERATION IS THE VALUE
!       ASSIGNED.  THUS,  A=B=C=1;    ASSIGNS 1 TO ALL THREE
!       VARIABLES.  THE EXAMPLES:  A=<T5>;
!       A=B=C=<T5>;     B=1+(A=B=5+3);          ARE CORRECT
!       BUT      A+1=3;  1+A=3;  ARE NOT.
!
!       "A;"  MAY BE USED TO PRINT THE VALUE OF A.
!
!       THREE STACKS ARE MAINTAINED IN THIS PROGRAM.
!       THE "MAIN STACK" IS  MAIN_STK AND ITS POINTER IS
!       MAIN_STK_POINTER.
!       ALL VALUES AND OPERATORS END UP ON IT IN RIGHT ORDER.
!
!       THE DERAILING STACK FOR OPERATORS IS OPERATOR_STACK.
!       OP_STACK_PTR IS ITS POINTER.
!       THIS STACK HOLDS LOWER PRECEDENCE OPERATORS AS HIGHER
!       PRECEDENCE OPERATORS ACCUMULATE.  THIS STACK IS EMPTIED
!       WHEN THE ";" IS PROCESSED.
!
!       THE EVALUATION STACK IS EVAL_STK.  ITS POINTER IS
!       EVAL_STK_PTR.
!       IT HOLDS OPERANDS WHILE THE MAIN STACK IS SCANNED FOR
!       OPERATORS. THE RESULTS OF OPERATIONS PERFORMED
!       GO ON THE EVALUATION STACK.  THIS STACK IS MANAGED
!       BY EVAL_POLISH AND ITS FRIENDS.
```

```
!
! TABLE OF CONTENTS:
!
FORWARD ROUTINE
!       MAINLOOP,                       ! MAIN PROGRAM LOGIC
        EXPRESSION,                     ! READ, PARSE, AND EVALUATE
                                        ! AN EXPRESSION
        INPUT_CYCLE,                    ! PARSE AN EXPRESSION
        PROCESS_OPR,                    ! PROCESS AN OPERATOR
        READ_UNTIL_DEL,                 ! LEXEME BUILDING
        GET_CHARACTER:NOVALUE,          ! GET A CHARACTER FROM INPUT
                                        ! STREAM
        PUSH_OPERATOR:NOVALUE,          ! PUSH ONTO OPERATOR STACK
        POP_OPERATOR,                   ! POP OFF OF OPERATOR STACK
        PUSH_MAIN_STACK:NOVALUE,        ! PUSH ONTO MAIN STACK
        POP_MAIN_STACK,                 ! POP OFF OF MAIN STACK
        EVAL_POLISH:NOVALUE,            ! EVALUATE A POLISH-TYPE
                                        ! EXPRESSION
        EVAL_OPERATOR,                  ! EVALUATE AN OPERATOR
        EVAL_ADDRESS,                   ! EVALUATE AN ADDRESS
        EVAL_VALUE                      ! EVALUATE A VALUE
        PUSH_EVAL_STACK:NOVALUE,        ! PUSH ONTO EVALUATION STACK
        POP_EVAL_STACK,                 ! POP OFF OF EVALUATION STACK
        PRINT_STRING:NOVALUE,           ! PRINT AN EXPRESSION
        PRINT_STACK,                    ! PRINT THE CONTENTS OF THE
                                        ! MAIN STACK
        ERROR;                          ! PRINT AN ERROR MESSAGE
!
!
! INCLUDE FILES:
!
REQUIRE
        'BLI:COMIOG.REQ';
!
! MACROS:
!
MACRO
        LEXEME_TYPE=    LEXEME[0]%,
        LEXEME_VALUE=   LEXEME[1]%,
        MAIN_TYPE=      MAIN_STK[.MAIN_STK_PTR-2]%,
        MAIN_VALUE=     MAIN_STK[.MAIN_STK_PTR-1]%,
        TOPOP=          OPERATOR_STACK[.OP_STACK_PTR-1]%;
```

```
!
! EQUATED SYMBOLS:
!
LITERAL
!
!
!                       OPERATOR CODES
!
        FIRST=          0,              ! "FIRST" OPERATOR
        OPEN_PAREN=     1,              ! OPEN PARENTHESIS
        CLOSE_PAREN=    2,              ! CLOSE PARENTHESIS
        MULTIPLY=       3,              ! MULTIPLICATION "*"
        PLUS=           4,              ! ADDITION "+"
        MINUS=          5,              ! SUBTRACTION "-"
        DIVIDE=         6,              ! DIVISION "/"
        SEMI_COL=       7,              ! SEMI-COLON ";"
        EQUAL=          8,              ! ASSIGNMENT "="
        NEGATIVE=       9,              ! NEGATION (UNARY MINUS) "-"
        CUTOFF=         2,              ! OPEN PAREN AND FIRST FOR NEG
!
!                       MAIN STACK ELEMENT CODES
!
        IS_NAME=        1,              ! VARIABLE NAME
        IS_DECIMAL=     2,              ! INTEGER VALUE
        IS_OPERATOR=    3,              ! OPERATOR
        IS_NONE=        0;              ! "NOTHING" (SPECIAL ELEMENT)
!
!                       PRECEDENCE TABLES
!
!
!        PRCDENCE_1 IS THE PRECEDENCE OF THE CURRENT OPERATOR
!        PRCDENCE_2 IS THE PRECEDENCE OF THE OPERATOR ON TOP OF
!        OP-STACK.
!        THE TWO ARE COMPARED IN INPUT_CYCLE.
!        THE TWO LISTS ARE BASICALLY THE SAME, BUT:
!
!        NOTE: PRECEDENCE_2 IS THE SAME AS PRECEDENCE_1 FOR ALL
!        OPERATORS EXCEPT "(" WHERE IT IS REDUCED TO 1 AND "=" WHERE
!        IT IS REDUCED TO 2. CLOSE PAREN WILL FORCE "=" ONTO
!        STACK AND WILL FORCE ALL OTHER OPERATORS DOWN TO "("
!        WHICH, WITH ")", IS REMOVED BY THE ACTION OF ")".
!
BIND
        PRCDNCE_1= UPLIT(0,9,2,7,4,5,6,1,3,8):VECTOR[10],
!                           ( ) * + - / ; = -
        PRCDNCE_2= UPLIT(0,1,2,7,4,5,6,1,2,8):VECTOR[10],
!
!                       ASCII VALUE OF OPERATORS
!
        OPNAMES=        PLIT(
                                PLIT (%ASCIZ 'FIRST'),
                                PLIT (%ASCIZ '('),
                                PLIT (%ASCIZ ')'),
                                PLIT (%ASCIZ '*'),
                                PLIT (%ASCIZ '+'),
                                PLIT (%ASCIZ '-'),
                                PLIT (%ASCIZ '/'),
                                PLIT (%ASCIZ ';'),
                                PLIT (%ASCIZ '='),
                                PLIT (%ASCIZ 'NEG')):VECTOR[50];
!
!                       SIZE PARAMETERS
!
LITERAL
        INPUT_SIZE=     133,                    ! INPUT AREA SIZE
        OUT_MSG_MAX=    132,                    ! MAX OUTLINE LENGTH
        STACK_SIZE=     400,                    ! SIZE OF STACKS
        STOR_LEN=       26,                     ! SIZE OF STORAGE
!
!                       MISC.
!
        NOTHING=        0,                      ! A CONDITION
        CAR_RETURN =    %O'15',                 !CARRIAGE RETURN
        CHARMASK =      (1^5)-1;                ! MASK LOW BITS


!
! OWN STORAGE
!
OWN
!                       STACKS
!
        MAIN_STK:       VECTOR[STACK_SIZE],     ! MAINSTACK
        OPERATOR_STACK: VECTOR[STACK_SIZE],     ! OPERATOR STACK
        EVAL_STK:       VECTOR[STACK_SIZE],     ! EVALUATION STACK
!
!                       STACK POINTERS
!
        MAIN_STK_PTR,
        OP_STACK_PTR,
        EVAL_STK_PTR,
!
!                       PARSING VARIABLES AND AREAS
!
        CHAR,                                   ! SINGLE ASCII
                                                ! CHARACTER INPUT
        STORAGE:        VECTOR[STOR_LEN],       ! IDENTIFIER VALUE
                                                ! STORAGE AREA
        DECVALUE,                               ! DECIMAL VALUE
        LEXEME:         VECTOR[2],              ! LEXICAL ELEMENT
                                                  ! LEXEME[0] = TYPE
                                                  ! LEXEME[1] = VALUE
        OPERATOR,                               ! OPERATOR CODE
                                                ! (SEE ABOVE)
        PAREN_LEVEL,                            ! PARENTHESES LEVEL
        INPUT:          VECTOR[INPUT_SIZE],     ! INPUT LINE
        INPUT_POINTER,                          ! INPUT LINE POINTER
        INPUT_LENGTH,                           ! LENGTH OF INPUT LINE
        ERRORV;
```

```
!
! EXTERNAL REFERANCES:
!
!       NONE
```

```
ROUTINE MAINLOOP  :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!
!       THIS IS THE MAIN ROUTINE OF THIS MODULE.  IT CONTAINS THE
!       GROSS LOGIC OF THE MODULE.
!       THE USER IS REPEATEDLY ASKED TO "TYPE EXPRESSION".  UPON
!       DOING SO THE EXPRESSION IS PARSED AND EVALUATED BY A
!       CALL TO THE ROUTINE EXPRESSION.
!       EXECUTION OF THIS ROUTINE (AND THE MODULE) IS HALTED BY
!       HITTING CONTROL C.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       STORAGE, INPUT, INPUT_LENGTH, INPUT_POINTER
!
! ROUTINE VALUE:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
!
!        RESET STORAGE TO ZERO VALUES
!
        INCR I FROM 0 TO (STOR_LEN - 1) DO
            STORAGE[.I]= 0;
!
! READ NEXT LINE
!
        WHILE 1 DO
            BEGIN
            TTY_PUT_CRLF();
            TTY_PUT_CHAR( %C'*' );                 ! PROMPT
            INCR I FROM 0 TO INPUT_SIZE-1
            DO
                BEGIN
                INPUT[.I] = TTY_GET_CHAR();
                IF .INPUT[.I] EQL CAR_RETURN    !CARRIAGE RETURN
                THEN
                    BEGIN
                    INPUT[.I] = %C';';  ! ONE EXTRA SEMICOLON
                    INPUT_LENGTH = .I;
                    EXITLOOP;
                    END;
                END;
            INPUT_POINTER = -1;
            IF EXPRESSION() THEN RETURN
            END;
        END;
```

```
ROUTINE EXPRESSION  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!
!       LOGICALLY,
!
!       THIS ROUTINE REPEATEDLY CALLS THE ROUTINE INPUT_CYCLE
!       IN ORDER TO READ AND PARSE THE EXPRESSION, PRINTS THE
!       EXPRESSION, PRINTS THE CONTENTS OF THE STACK JUST BUILT, AND
!       THEN EVALUATES THE EXPRESSION VIA A CALL TO EVAL_POLISH.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       PAREN_LEVEL
!
! COMPLETION CODES:
!
!       RETURNED AS ROUTINE VALUE;
!               0 - NO ERRORS REPORTED
!               1 - ERROR ENCOUNTERED
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
                CONDITION;      ! VALUE RETURNED BY INPUT_CYCLE

        PAREN_LEVEL = 0;
        DO
            CONDITION = INPUT_CYCLE()
        UNTIL .CONDITION NEQ 0;
        IF .CONDITION  EQL 1 THEN  RETURN 1;            !ERROR
        PRINT_STRING();
        PRINT_STACK();
        EVAL_POLISH();
        RETURN 0
        END;
```

```
ROUTINE INPUT_CYCLE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!
!       THIS ROUTINE MAKES CALLS TO ROUTINE READ_UNTIL_DELITER
!       ACCESSING LEXEMES AND DELIMITERS.  BASED ON THE TYPE
!       THE ROUTINE PERFORMS VARIOUS FUNCTIONS.
!       NOTE:
!               THERE IS AN INTERNAL ROUTINE CALLED PROCESS_OPR, WHICH
!       HANDLES OPERATOR DELIMITERS.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       LEXEME_TYPE, LEXEME_VALUE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! COMPLETION CODES:
!
!       RETURNED AS ROUTINE VALUE;
!               0 - NO ERRORS ENCOUNTERED
!               1 - ERROR ENCOUNTERED
!               2 - END OF EXPRESSION
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
                VALUE;                  ! VALUE TO BE RETURNED

        IF READ_UNTIL_DEL() THEN RETURN 1;
        IF .LEXEME_TYPE NEQ IS_NONE
        THEN
            BEGIN
            PUSH_MAIN_STACK(.LEXEME_TYPE);
            PUSH_MAIN_STACK(.LEXEME_VALUE)
            END
        ELSE                        !UNARY OPERATOR
            IF (.OPERATOR NEQ MINUS AND
                 .OPERATOR NEQ OPEN_PAREN)
            THEN
                RETURN(ERROR(4))
            ELSE
                IF .OPERATOR EQL MINUS
                THEN
                    IF .PRCDNCE_2[.TOPOP] LSS CUTOFF
                    THEN
                        OPERATOR = NEGATIVE
                    ELSE
                        RETURN(ERROR(5));
        PROCESS_OPR
        END;
```

```
ROUTINE PROCESS_OPR =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE HANDLES OPERATORS (DELIMITERS).  IT
!       KEEPS TRACK OF THE PARENTHESES COUNT AND THE PROPER
!       SYNTAX OF EXPRESSIONS.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       OPERATOR, PAREN_LEVEL, TOPOP, LEXEME_TYPE,
!       PRECIDENCE_1, PRECIDENCE_2
!
! IMPLICIT OUTPUTS:
!
!       PAREN_LEVEL
!
! COMPLETION CODES:
!
!       RETURNED AS ROUTINE VALUE;
!               0 - NO ERRORS ENCOUNTERED
!               1 - ERROR ENCOUNTERED
!               2 - END OF EXPRESSION
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
                CONDITION;      ! VALUE RETURNED BY PROCESS_OPR

        IF .OPERATOR EQL OPEN_PAREN
        THEN
            PAREN_LEVEL = .PAREN_LEVEL+1;

        IF .OPERATOR EQL CLOSE_PAREN
        THEN
            PAREN_LEVEL = .PAREN_LEVEL-1;

        WHILE .PRCDNCE_1[.OPERATOR] LEQ .PRCDNCE_2[.TOPOP]
        DO
            BEGIN
            PUSH_MAIN_STACK(IS_OPERATOR);
            PUSH_MAIN_STACK(POP_OPERATOR());
```

```
    END;

IF .OPERATOR EQL SEMI_COL
THEN
    IF .PAREN_LEVEL EQL 0
    THEN
        RETURN 2
    ELSE
        RETURN (ERROR(9));

IF .OPERATOR EQL CLOSE_PAREN
THEN
    BEGIN
    IF .TOPOP NEQ OPEN_PAREN THEN  RETURN(ERROR(3));
    POP_OPERATOR();
    IF READ_UNTIL_DEL() THEN  RETURN 1;
    IF .LEXEME_TYPE NEQ IS_NONE THEN RETURN(ERROR(6));
    CONDITION=PROCESS_OPR();
    IF .CONDITION GTR 0
    THEN
        RETURN .CONDITION;
    END
ELSE
    PUSH_OPERATOR(.OPERATOR);
RETURN 0

END;
```

```
ROUTINE READ_UNTIL_DEL  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE DOES THE ACTUAL PARSING OF THE INPUT EXPRESSION
!       LOOKING FOR SYMBOLS, NUMBERS AND OPERATORS(DELIMITERS).
!       IT ALWAYS ATTEMPTS TO RECOGNIZE AN OPERATOR AND
!       RETURN ITS CODE.
!       PRIOR TO SEARCHING FOR THE OPERATOR IT LOOKS FOR A SYMBOL
!       (IS_NAME) OR INTEGER(IS_DECIMAL).  IF NONE OF THESE
!       ARE FOUND THEN IS_NONE IS RETURNED IN THE GLOBAL VARIABLE
!       LEXEME_TYPE.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!        CHAR, OPERATOR
!
! IMPLICIT OUTPUTS:
!
!       OP_STACK_PTR, MAIN_STK_PTR, ERRORV, LEXEME_TYPE,
!       LEXEME_VALUE, DECVALUE, OPERATOR
!
! ROUTINE VALUE:
!
!       OPEN_PAREN, CLOSE_PAREN, MULTIPLY, PLUS, MINUS
!       DIVIDE, SEMI_COL, EQUAL, ERROR(1), ERROR(2)
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        IF .INPUT_POINTER EQL -1
        THEN
!
!           IF FIRST TIME THROUGH PLACE SPECIAL "FIRST" DELIMITER
!           ON THE MAIN_STK
!
            BEGIN
            GET_CHARACTER();
            OP_STACK_PTR = 0;
            MAIN_STK_PTR = 0;
            ERRORV = 0;
            PUSH_OPERATOR(FIRST);                   !INITDEL
            END;
```

```
!       FIRST SEARCH FOR A SYMBOL OR INTEGER

        LEXEME_TYPE = IS_NONE;  !FOR THERE MAY NOT BE ONE
        IF (.CHAR GEQ %C'A' AND .CHAR LEQ %C'Z') OR
           (.CHAR GEQ %C'a' AND .CHAR LEQ %C'z')
        THEN
            BEGIN                             !CONVERT CHAR TO AN
            LEXEME_TYPE = IS_NAME;            !INDEX INTO STORAGE
                                              !ARRAY
            LEXEME_VALUE = (.CHAR AND CHARMASK)-1;
            GET_CHARACTER()
            END
        ELSE
            IF (.CHAR GEQ %C'0' AND .CHAR LEQ %C'9')
            THEN
                BEGIN
                DECVALUE = 0;
                WHILE (.CHAR GEQ %C'0' AND .CHAR LEQ %C'9') DO
                    BEGIN
                    DECVALUE = 10*.DECVALUE+.CHAR-%C'0';
                    GET_CHARACTER();
                    END;
                LEXEME_TYPE = IS_DECIMAL;
                LEXEME_VALUE = .DECVALUE;     !DECIMAL INTEGER VALUE
                END;

!       NOW GET DELIMITER WHETHER OR NOT WE HAD AN IDENTIFIER OR NUMBER

        IF (.CHAR LSS %C'(' OR .CHAR GTR %C'=')
        THEN
                RETURN(ERROR(1))
        ELSE
            BEGIN
            OPERATOR=
                (CASE (.CHAR) FROM %C'(' TO %C'=' OF
                    SET
                    [%C'(']:
                        OPEN_PAREN;
                    [%C')']:
                        CLOSE_PAREN;
                    [%C'*']:
                        MULTIPLY;
                    [%C'+']:
                        PLUS;
                    [%C'-']:
                        MINUS;
                    [%C'/']:
                        DIVIDE;
                    [%C';']:
                        SEMI_COL;
                    [%C'=']:
                        EQUAL;
                    [INRANGE]:
                        ! ALL OTHER VALUES ARE IN ERROR
                        RETURN(ERROR(2))
                    TES);
            GET_CHARACTER();
            IF .OPERATOR EQL 0 THEN RETURN(ERROR(2))
             END;
        END;
```

```
ROUTINE GET_CHARACTER  :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTIION:
!
!       THIS ROUTINE ACCESES THE NEXT CHARACTER FROM THE INPUT STREAM
!       AND PLACES IT IN THE GLOBAL VARIABLE CHAR.
!       ALL CHARACTERS WITH AN OCTAL VALUE LESS THAN 40 ARE IGNORED.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       INPUT_POINTER
!
! IMPLICIT OUTPUTS:
!
!       INPUT_POINTER, CHAR
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        DO
            BEGIN
            INPUT_POINTER = .INPUT_POINTER + 1;
            CHAR = .INPUT[.INPUT_POINTER];
            END
        UNTIL (.CHAR GTR %C' ');
        END;
```

```
ROUTINE PUSH_OPERATOR(ELEMENT) :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE "PUSHES" AN ELEMENT ONTO THE OPERATOR_STACK.
!
! FORMAL PARAMETERS:
!
!       ELEMENT - OPERATOR TO BE ADDED TO STACK
!
! IMPLICIT INPUTS:
!
!       OP_STACK_PTR
!
! IMPLICIT OUTPUTS:
!
!       OP_STACK_PTR, OPERATOR_STACK
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        OPERATOR_STACK[.OP_STACK_PTR] = .ELEMENT;
        OP_STACK_PTR = .OP_STACK_PTR+1
        END;
```

```
ROUTINE POP_OPERATOR  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!        THIS ROUTINE "POPS" A DATA ELEMENT OFF OF THE OPERATOR_STACK.
!
! FORMAL PARAMETERS:
!
!        NONE
!
! IMPLICIT INPUTS:
!
!        OP_STACK_PTR, OPERATOR_STACK
!
! IMPLICIT OUTPUTS:
!
!        OP_STACK_PTR
!
! ROUTINE VALUE:
!
!        VALUE OF ELEMENT POPPED FROM STACK
!
! SIDE EFFECTS:
!
!        NONE
!
!--

        BEGIN
        OP_STACK_PTR = .OP_STACK_PTR-1;
        .OPERATOR_STACK[.OP_STACK_PTR]
        END;
```

```
ROUTINE PUSH_MAIN_STACK(ELEMENT) :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE WILL "PUSH" AN ELEMENT ONTO THE MAIN_STK.
!
! FORMAL PARAMETERS:
!
!       ELEMENT - DATA TO BE PUSHED ON MAIN_STK
!
! IMPLICIT INPUTS:
!
!       MAIN_STK_PTR
!
! IMPLICIT OUTPUTS:
!
!       MAIN_STK, MAIN_STK_PTR
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        MAIN_STK[.MAIN_STK_PTR]= .ELEMENT;
        MAIN_STK_PTR= .MAIN_STK_PTR+1
        END;
```

```
ROUTINE POP_MAIN_STACK  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE "POPS" AN ELEMENT OFF OF THE MAIN_STK
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       MAIN_STK_PTR, MAIN_STK
!
! IMPLICIT OUTPUTS:
!
!       MAIN_STK
!
! ROUTINE VALUE:
!
!       VALUE OF ELEMENT POPPED FROM THE STACK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        MAIN_STK_PTR= .MAIN_STK_PTR-1;
        .MAIN_STK[.MAIN_STK_PTR]
        END;
```

```
ROUTINE EVAL_POLISH  :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTIION:
!
!       THIS ROUTINE DOES THE ACTUAL EVALUATION OF EXPRESSION
!       WHICH HAS NOW BEEN PARSED AND RESIDES ON THE MAIN_STK.
!       OPERANDS (VARIABLES AND INTEGERS) ARE SHUNTED OFF AND PLACED
!       ONTO THE EVAL_STK.
!       OPERATORS ARE EVALUATED BY MAKING A CALL TO EVAL_OPERATOR.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       MAIN_STK_PTR, MAIN_STK, EVAL_STK
!
! IMPLICIT OUTPUTS:
!
!       EVAL_STK_PTR, LEXEME_TYPE, LEXEME_VALUE, EVAL_STK
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        EVAL_STK_PTR = 0;
        INCR I FROM 0 TO .MAIN_STK_PTR-1 BY 2 DO
            BEGIN
            LEXEME_TYPE = .MAIN_STK[.I];
            LEXEME_VALUE = .MAIN_STK[.I+1];
            IF .LEXEME_TYPE NEQ IS_OPERATOR
            THEN
                BEGIN
                PUSH_EVAL_STACK(.LEXEME_TYPE);
                PUSH_EVAL_STACK(.LEXEME_VALUE)
                END
            ELSE
                EVAL_OPERATOR(.LEXEME_VALUE);
            END;
        IF .MAIN_STK_PTR EQL 2
        THEN
            EVAL_STK[1] = EVAL_VALUE(); ! THE CASE "A;"
        TTY_PUT_CRLF();
        TTY_PUT_QUO('VAL:        ');
        TTY_PUT_INTEGER(.EVAL_STK[1],10,10);
        END;
```

```
ROUTINE EVAL_OPERATOR(STACK_OPERATOR) =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE EVALUATES THE OPERATOR STACK_OPERATOR.
!       THE PROPER NUMBER OF OPERANDS ARE ACCESSED FORM THE MAIN_STK.
!       AFTER EVALUATION THE VALUE IS PLACED ON THE EVAL_STK.
!
! FORMAL PARAMETERS:
!
!       STACK_OPERATOR - OPERATOR TO BE EVALUATED
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       STORAGE
!
! ROUTINE VALUE:
!
!       ERROR(3)
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
            VALUE_1,                                ! INTERMEDIATE
                                                    ! SAVE AREAS
            VALUE_2,                                ! ...
            VALUE_3;                                ! ...

        VALUE_3 =
             (SELECT .STACK_OPERATOR OF
                 SET

                 [ALWAYS]:

!                    DO THIS FIRST - DETERMINE THE NUMBER OF OPERANDS
!                    NEEDED BY THIS PARTICULAR OPERATOR

                     BEGIN
                     VALUE_2 =  EVAL_VALUE();
                     VALUE_1 =
                         (IF .STACK_OPERATOR EQL EQUAL THEN
                             EVAL_ADDRESS()
```

```
                        ELSE
                            IF .STACK_OPERATOR NEQ NEGATIVE THEN
                                EVAL_VALUE())
                    END;

                [NEGATIVE]:

!                   NEGATION - (UNARY MINUS)
                    -.VALUE_2;


                [MULTIPLY]:
                    .VALUE_1 * .VALUE_2;

                [DIVIDE]:
                    .VALUE_1 / .VALUE_2;

                [MINUS]:
                    .VALUE_1 - .VALUE_2;

                [PLUS]:
                    .VALUE_1 + .VALUE_2;

                [EQUAL]:
                    ! STORE THE VALUE IN VALUE_2
                    STORAGE[.VALUE_1] =  .VALUE_2;

                [OTHERWISE]:
                    RETURN(ERROR(8));

                TES);
        PUSH_EVAL_STACK(IS_DECIMAL);
        PUSH_EVAL_STACK(.VALUE_3);
        END;
```

```
ROUTINE EVAL_ADDRESS  =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE IS CALLED WHEN THE ASSIGNMENT OPERATOR IS TO BE
!       EVALUATED.  THE VALUE RETURNED IS THE ADDRESS (INDEX) OF THE
!       IDENTIFIER IN STORAGE.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
!
!       ERROR(7), ADDRESS(INDEX) OF THE IDENTIFIER FROM THE
!       TOP OF EVAL_STK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
            VAL,                                ! TEMPORARY VALUE
            TYPE;                               ! TEMPORARY TYPE

        VAL = POP_EVAL_STACK();
        TYPE = POP_EVAL_STACK();
        IF .TYPE NEQ IS_NAME THEN  RETURN(ERROR(7));
        .VAL
        END;
```

```
ROUTINE EVAL_VALUE =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE ACESSES THE VALUE OF THE IDENTIFIER.  THE
!       EVAL_STK VALUE IS USED TO INDEX THE IDENTIFIER VALUE STORAGE
!       AREA (STORAGE).
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       STORAGE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
!
!       VALUE OF THE IDENTIFIER ON THE TOP OF EVAL_STK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        LOCAL
            TYPE,                               ! TEMPORARY TYPE
            VAL;                                ! TEMPORARY VALUE
        VAL = POP_EVAL_STACK();
        TYPE = POP_EVAL_STACK();
        IF .TYPE EQL IS_NAME THEN  VAL = .STORAGE[.VAL];
        .VAL
        END;
```

```
ROUTINE PUSH_EVAL_STACK(ELEMENT) :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION
!
!       THIS ROUTINE "PUSHES" A DATA ELEMENT ONTO THE EVAL_STK.
!
! FORMAL PARAMETERS:
!
!       ELEMENT - DATA TO BE PLACED ON EVAL_STK
!
! IMPLICIT INPUTS:
!
!       EVAL_STK, EVAL_STK_PTR
!
! IMPLICIT OUTPUTS:
!
!       EVAL_STK_PTR
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        EVAL_STK[.EVAL_STK_PTR] = .ELEMENT;
        EVAL_STK_PTR = .EVAL_STK_PTR+1
        END;
```

```
ROUTINE POP_EVAL_STACK =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE "POPS" AN ELEMENT OFF OF THE EVAL_STK.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       EVAL_STK_PTR, EVAL_STK
!
! IMPLICIT OUTPUTS:
!
!       EVAL_STK_PTR
!
! ROUTINE VALUE:
!
!       VALUE POPPED FROM EVAL_STK
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        EVAL_STK_PTR = .EVAL_STK_PTR-1;
        .EVAL_STK[.EVAL_STK_PTR]
        END;
```

```
ROUTINE PRINT_STRING :NOVALUE =

!++
!
! FUCTIONAL DESCRIPTION
!
!       THIS ROUTINE PRINTS OUT THE EXPRESSION JUST READ IN.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       INPUT
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        TTY_PUT_CRLF();
        INCR I FROM 0 TO .INPUT_LENGTH-1 DO
            BEGIN
            IF .INPUT[.I] EQL CAR_RETURN
            THEN
                EXITLOOP;
            TTY_PUT_CHAR(.INPUT[.I]);
            END;
        END;
```

```
ROUTINE PRINT_STACK =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE PRINTS OUT THE CONTENTS OF MAIN_STK IN SYMBOLIC
!       FORMAT.
!
! FORMAL PARAMETERS:
!
!       NONE
!
! IMPLICIT INPUTS:
!
!       MAIN_STK_PTR, MAIN_STK
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
! COMPLETION CODES:
!
!       NONE
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        INCR I FROM 0 TO .MAIN_STK_PTR-1 BY 2 DO
            BEGIN
            TTY_PUT_CRLF();
            SELECT .MAIN_STK[.I] OF
                SET

                [IS_NAME]:
                    TTY_PUT_CHAR(.MAIN_STK[.I+1] + %C'A');

                [IS_DECIMAL]:
                    TTY_PUT_INTEGER(.MAIN_STK[.I+1],10,10);

                [IS_OPERATOR]:
                    TTY_PUT_ASCIZ(.OPNAMES[.MAIN_STK[.I+1]]);

                TES;
            END;
        END;
```

```
ROUTINE ERROR(ERROR_NUMBER) =

!++
!
! FUCTIONAL DESCRIPTION:
!
!       THIS ROUTINE PRINTS OUT ERROR MESSAGES BASED ON THE
!       ERROR_NUMBER PASSED TO IT.  IT ALSO DUMPS THE CONTENTS OF THE
!       MAIN_STK AND PRINTS THE EXPRESSION IN ERROR.
!
! FORMAL PARAMETERS:
!
!       ERROR_NUMBER - INDEX INTO ERROR MESSAGE PLIT
!
! IMPLICIT INPUTS:
!
!       ERROR_MESSAGE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! ROUTINE VALUE:
!
!       1
!
! SIDE EFFECTS:
!
!       NONE
!
!--

        BEGIN
        MACRO
                MESSAGE(ARGUMENT) = PLIT (%ASCIZ ARGUMENT)%;

        BIND
                ERROR_MESSAGE = PLIT(
                    MESSAGE('ERR:0       NONE'),
                    MESSAGE('ERR:1       ILLEGAL CHARACTER ON INPUT'),
                    MESSAGE('ERR:        OPR EXPECTED, NOT FOUND'),
                    MESSAGE('ERR:3       EXCESS CLOSE PAREN'),
                    MESSAGE('ERR:4       ILLEGAL UNARY OPERATOR'),
                    MESSAGE('ERR:5       ILLEGAL USE OF UNARY MINUS'),
                    MESSAGE('ERR:6       OPERATOR MUST FOLLOW ")"'),
                    MESSAGE('ERR:7       ASSIGNMENT TO NON VARIABLE'),
                    MESSAGE('ERR:8       BAD OPERATOR ON STACK'),
                    MESSAGE('ERR:9       EXCESS OPEN PAREN'),
                    MESSAGE('ERR:10      NONE')
                                ):VECTOR[50];

        TTY_PUT_CRLF();
        TTY_PUT_MSG(.ERROR_MESSAGE[.ERROR_NUMBER],OUT_MSG_MAX);
```

```
         PRINT_STACK();
         PRINT_STRING();
         RETURN 1
         END;
END
ELUDOM
```

[End of Appendix C]

INDEX