TECHNICAL MEMORANDUM

120

Standards for software development and maintenance

J.K. Gibson

Operations Department

August 1986

European Centre for Medium-Range Weather Forecasts
Europäisches Zentrum für mittelfristige Wettervorhersage
Centre européen pour les prévisions météorologiques à moyen

1. INTRODUCTION

Maintaining software is usually more expensive in terms of manpower resources than creating software. In ECMWF's operational environment

- changes to improve the analysis, the forecast, or the quality of disseminated products must be made with a minimum of delay;
- computer hardware systems have a shorter life than the software applications they support;
- applications analysts and programmers are limited in number and there is a high staff turnover;
- many applications are of interest to Member States users, and to potential users external to ECMWF.

The costs of software maintenance have been well documented (e.g. Martin and McClure, 1983). Attention to programme methodology focused initially on modularity, then graduated to structured programming (Dijkstra, 1976). Conventions related to programming style have been used, often to good effect (Roberts, 1974; Frank, 1971; Gibson, 1980 and 1982). Nevertheless, maintenance demands, and will continue to require considerably more resources than the seemingly more creative task of software development!

2. SOFTWARE DEVELOPMENT AND MAINTENANCE

2.1 Basic Concepts

The cost of software maintenance depends on a number of factors. These include:

- the software quality - is it error free?
- the software design - does it do its job?
- the code complexity - can it be understood?
- the documentation - is there any?
- the data interface - is it flexible?
- the need to modify - can all future requirements ever be foreseen?
- analyst availability - are they fixing other bugs?

Reduction of these costs requires careful planning, a methodological approach, and a set of acceptable standards.

2.2 Aims of the Standards

It is essential that software for the second generation ECMWF Meteorological Operational System (EMOS II) should be:

- well planned and designed
- well written
- sufficiently documented
- maintainable
- transportable
- efficient
- flexible.

The following development standard, coding standard and maintenance standard are attempts to satisfy these aims. They are presented in the belief that, where applied, they will contribute to the development and modification of systems that will be maintainable, and that the methodology proposed will reduce the costs of software maintenance.

3. A SOFTWARE DEVELOPMENT STANDARD

3.1 Specification and Design

A well thought-out programme is less difficult to code, produces fewer bugs, and is often readily maintainable. Thus it is essential that sufficient attention be given to specifying the tasks to be accomplished, and designing the means by which they shall be achieved. For each significant programme or sub-system within EMOS II the specification and design should be presented as a Technical Memorandum.

First, the objectives of the system should be considered. These should not be confined to immediate objectives - desirable future enhancements should be anticipated and planned wherever possible.

Next, the components of the system should be planned. These should be designed in a modular fashion, so that modules are interrelated in hierarchical, tree-like structures. Modules should not be allowed to become too complex, as this will increase the difficulty of maintenance, as well as making the module difficult to test. Relationships between modules should be kept simple, further facilitating testing, modification and maintenance.

High level languages should be used in preference to assembler languages, and appropriate use should be made of standard, well tested libraries. Wherever possible use of the high level language should be confined to the ANSI sub-set. Non-standard features, if necessary, should be confined to separate sets of modules to improve portability.

Attention should be given to the dataflow through and within the system. Data structures should be kept simple. Standard data forms should be used where appropriate. Data input, output and data manipulation should be confined to a set of data handling modules, separated from the applications modules. This will reduce machine dependence and enhance transportability, since only the data handling modules will need alteration to migrate to a different machine environment.

The problem of the relationship between physical and logical data representation should be considered. So too should the problem of data location. Where possible, data formats should conform to machine independent, standard forms. Files should be grouped into data bases. Intermediate software should retrieve or add data, performing any required reformating or data conversion at the same time. In this way the applications will not be closely coupled to the file structure or the data storage format, enabling much greater flexibility at a later date.

Consideration should be given at the design stage to

- how the system should be tested
- how the system could be modified
- how the system will be maintained.

If step by step testing of individual modules, related sets of modules, and major sub-components is planned in advance, check-out of the system will be both comprehensive and readily achieved. Planning for future modification should enable such modifications to be made with a minimum of disturbance to the system as a whole. Planning for future maintenance may include reviewing which modules would need modification to migrate the system to a different computer.

Past experience at ECMWF has shown that good design documentation is essential, not least for future maintenance. Systems can best be maintained if the maintenance analyst understands not just the code, but the design philosophy that resulted in the code being written.

Where the specification and design document also includes plans for testing, future modification and maintenance, its value is greatly enhanced.

3.2 Coding

Coding should begin only when the design document has been completed, discussed, modified and approved. If the design has been sufficiently thorough, the design document will provide a sufficient specification. If changes are considered desirable then the proposed changes will require discussion, followed by subsequent amendments to the design document.

The coding standard contained in section 4 includes provision for 3 levels of documentation within the source code - overview, external and internal. Overview documentation indicates the purpose of each routine. This should follow directly from the specification of the modules within the design document. Routines should perform a single, logical sub-task. Calling sequences should be such that routines are related in a tree-like, hierarchical manner. Applications routines should call separate routines to perform input, output, data manipulation, or any machine or operating system dependent function. The only input/output handled directly by applications routines should be the writing of messages to an output file.

External documentation consists of the description of each routine placed at its head. It is recommended that the overview and external documentation be coded for the complete system before coding the executable statements. This results in the creation of a clear picture of the functions to be coded before attacking

the programming problem. The external documentation provides details of the interface, the method, the externals to be called, the existence of further documentation, the author, and the modification record.

Internal documentation serves two purposes - it relates the code to supplementary documentation, and it indicates, sub-section by sub-section, the purpose of the code. Source code which is interspersed with many comments is difficult to follow, difficult to understand, and thus difficult to maintain. If the documentation at the head of a routine is well written, then one line headings at the top of each sub-section of 10 lines or so of code is sufficient.

Coding should be such that the source code is easy to follow. Structured techniques should be used, with indentation of IF clauses, and avoidance of branching where possible. Where branches are necessary, forward branches should be considered preferable to backward branches. Backward branches should be used only when there is no reasonable alternative. Subroutine complexity should not be excessive. Each routine should contain less than 300 executable statements. Using the following measure of complexity:

- each executable statement - score 1
- each subroutine call - score 5
- each branch - score 10

each routine should have a maximum score of 400.

3.3 Testing

As each routine is written, it should be reviewed. Code reviews should involve at least two analysts. The following should be considered:

- does the source code satisfy the standards?
- can the source code satisfy a standard ANSI compiler?
- is the code easy to understand - can 90% of it be understood after studying it for 10 minutes?
- is the code too complex? (see 3.2 above)
- is the interface to the routine simple and sufficient?
- does the routine produce the required results?

- can modifications be made if required?
- are error situations detected and correctly treated?
- is the routine efficient?

Having tested individual routines, sub-sets of the system should be integrated in a demonstrable and testable manner. Such sub-sets should in turn be reviewed. Finally, the total system should be assembled and run in quasi-operational mode (i.e. along side the current operational systems). Only when results of this final integrated test have been assessed should the new system be accepted for operational use.

4. A CODING STANDARD

4.1 Introduction

The coding style used for EMOS has been generally good. The level of comments has been about right, and sensible use has been made of structured techniques. Nevertheless, the advantages to be gained from moving to a slightly more formal standard would be, in the author's opinion, well worth the effort required. The standard described below is based on the DOCTOR system (Gibson, 1980 and 1982), which in turn has much of its origins in a CDC concept (Frank, 1971), and in the OLYMPUS system (Roberts, 1974). DOCTOR has already been used as the standard for two major systems - the analysis and the forecast. The description which follows contains some further modifications which seem relevant to operational practice, and, in particular, to the predominant use of Fortran 77 for new systems.

4.2 Aims of the Standard

The purpose of a coding standard is to produce a coding style which leads to well presented source code: such code should be well structured, well documented and thus easy to understand. In addition, DOCTOR aims to provide standards for the recognition of the scope of variables, partitioning of the code within routines, and a means of extracting documentation from the source code. To sum up, the basic aims are:

- to provide well presented code;
- to produce code which can be easily understood;
- to enable the inclusion of extractable documentation;

- to facilitate recognition of variable types and their scope (global, local, arguments, etc.);
- to set up points of reference in an orderly way;
- to establish conventions concerning modularily, structure, linkage, and style with future maintainability in mind.

4.3 Coding Conventions

Code should follow a modular structure, each module fulfilling a stated purpose. Modules should not become too complex - a maximum complexity score of 200 based on the criteria given in 3.2 above is recommended. Modules should have a hierarchical structure within a programme, and only call other modules at a lower level.

Each module should contain one entry point, and at most two exits (a normal return, and an abnormal end). The entry point should be at the head of the routine, and the exit sequence at the base (see figure 4.1). RETURN should not be coded elsewhere in the body of the routine. Exit from a module should be via the normal return (with a return code if necessary) or by means of a deliberate abort.

```
ENTRY POINT                )..at head of routine
   ...
(routine body)
   ...
IF (HARD ERROR) THEN       )
     abort programme       )
ELSE                       )   return sequence at base of
       RETURN              )   routine
ENDIF                      )
END                        )
```

Figure 4.1 Entry and Return conventions

Error handling should follow the following conventions:

- on error detection, a sensible message which includes the error location should be written to the output file;
- hard or soft response to error detection should be signalled by a return code argument. This argument should be passed as zero to indicate hard response, positive to indicate soft response;
- a separate return code should be allocated for each detectable error condition;
- the return code should be returned as zero if no error is detected, or set to indicate the error(s) encountered if soft response is requested;
- if hard response is requested, the programme should be abnormally terminated on error detection, after writing the error report;
- STOP should never be used.

Separate modules should be coded to perform all input/output other than the output of messages to the output file. This makes it possible completely to change file formats and input/output methods without changing the main body of the code. The operational data bases should be used wherever possible.

All machine dependent or operating system dependent features should be avoided unless it is essential that they be used. Wherever such features are used, they should be isolated within separate modules kept in libraries apart from the main body of the code. This isolates the dependent features to collections of modules which need to be changed if the programme is to migrate to another environment. The main body of the code is then portable.

Names, subscripts, etc., should be as meaningful as possible, and should follow the conventions contained in 4.5 below.

GO TO should be avoided where possible. If branches are necessary, they should be confined to forward branches.

IF blocks should be indented. Nesting of IF blocks should never be more than 3 deep - deeper nesting destroys the understandability of the code.

DO loop boundaries should stand out. This can be achieved by commenting a blank line before and after the DO statement and the terminator statement. Alternatively, indentation may be used. Each Do loop should have a separate terminator.

Labels should never be associated with executable statements (e.g. in FORTRAN the non-executable CONTINUE should always be used). This enables labels to be moved without disturbing executable code. FORTRAN label numbers should be assigned in accordance with 4.4 below.

4.4. Coding Style

Each module should begin with a set of principle comments. These should contain:

- a title
- the PURPOSE of the routine
- the INTERFACE details, details of input/output parameters, arguments, etc.
- the EXTERNALS or other routines called
- a REFERENCE to further documentation
- the AUTHOR and date the routine was written, together with MODIFICATION details.

The code body of the module should be split into sections and sub-sections. Sections should be numbered (1 to n), as should be sub-sections within sections (1.1, 1.2 ... 1.m, 2.1, etc.). Thus, the M^{th} sub-section of the N^{th} section is numbered N.M. Each section should be separated from the previous section by a comment card containing minus signs or underlines in columns 7 to 72.

Each section should begin with:

- a section number and title (underlined)
- a CONTINUE statement numbered N00 (for the N^{th}section),

e.g.

```
C
C     ------------------------------------------------------
C*          1.   SET INITIAL VALUES.
C                ---  -------  -------
      100 CONTINUE
```

Each sub-section should begin with

- a sub-section number and title
- a CONTINUE statement numbered NM0 (Section N, sub-section M).

Numeric labels (e.g. FORTRAN statement numbers) should be related to the section and sub-section numbers. Thus section 3 begins with label 300; sub-section 12.1 begins with label 1210. This scheme allows a maximum of 10 labels per sub-section. Code requiring more than 10 labels per sub-section should be re-structured into sub-sections requiring less than 10 labels.

All branches in the code, including use of END=..., ERR=... in input/output statements, must be commented. The comments should refer to the section and sub-section branched to, not to the label (since label numbers or names do NOT appear in extracted documentation).

FORMAT statements should normally be grouped together, near the base of the subroutine. They should be labelled with numeric labels in the range 9000 to 9999. FORMAT descriptors may be used as an alternative to FORMAT statements if this aids readability of the source code.

The example subroutine in Fig. 4.2 illustrates the coding style.

4.5 Naming Convections

The purpose of the following naming conventions is to convey, through prefix letters, the type, scope, and nature of all variables within the programme. For the purpose of this standard minor changes have been made to the previous definition of the DOCTOR system (Gibson, 1982). These reflect

- the increased use of the facilities of FORTRAN 77, especially CHARACTER type
- rationalisation in the light of experience
- the desirability to restrict convention prefixes to a single letter where possible.

The type of a variable is indicated by the first letter of the variable's name according to Fig. 4.3.

```
      SUBROUTINE EXAMPLE(PA,KLEN,KDV)
C
C**** *EXAMPLE* - ROUTINE TO DEMONSTRATE THE *DOCTOR* STYLE.
C
C     PURPOSE.
C     --------
C
C          THE MAIN PURPOSE OF THIS ROUTINE IS TO DEMONSTRATE THE
C     CONVENTIONS, STYLE, AND PRESENTATION OF SOURCE CODE USING A
C     CODING STANDARD BASED ON THE *DOCTOR* SYSTEM.
C
C          ADDITIONALLY, THIS ROUTINE PRINTS THE MAXIMUM AND MINIMUM
C     VALUES OF AN ARRAY.
C
C**   INTERFACE.
C     ----------
C
C          *CALL* *EXAMPLE(PA,KLEN,KDV)*
C
C               *PA*      -  ARRAY TO BE EXAMINED (INPUT)
C               *KLEN*    -  LENGTH OF *PA.*
C               *KDV*     -  LOGICAL UNIT FOR OUTPUT MESSAGES.
C
C     METHOD.
C     -------
C
C          *PA* IS SCANNED, AND THE MINIMUM AND MAXIMUM VALUES
C     EXTRACTED. THESE ARE THEN WRITTEN TO FILE *KDV.*
C
C     EXTERNALS.
C     ----------
C
C          NONE.
C
C     REFERENCE.
C     ----------
C
C          NONE.
C
C     AUTHOR.
C     -------
C
C          J. K. GIBSON      *ECMWF*      14/05/86.
C
C     MODIFICATIONS.
C     --------------
C
C          NONE.
C
      IMPLICIT LOGICAL(L,O,G), CHARACTER*8(C,H,Y)
C
      DIMENSION PA(*)
C
C     ------------------------------------------------------------------
C
C*         1.      SET INITIAL VALUES.
C                   --- ------- ------
C
  100 CONTINUE
      ZMAX=-10.E20
      ZMIN= 10.E20
C
C     ------------------------------------------------------------------
C
```

Fig. 4.2: Sample routine illustrating coding style

Fig. 4.2 cont.

```
C*          2.      EXTRACT MAXIMUM AND MINIMUM.
C
C
  200 CONTINUE
C
      DO 212 J=1,KLEN
      IF (PA(J).LT.ZMIN) THEN
         ZMIN=PA(J)
         IMIN=J
      ENDIF
C
      IF (PA(J).GT.ZMAX) THEN
         ZMAX=PA(J)
         IMAX=J
      ENDIF
C
  212 CONTINUE
C
C     ------------------------------------------------------------------
C
C*          3.      PRINT RESULTS.
C
C
  300 CONTINUE
C
C*          3.1     PRINT HEADING.
  310 CONTINUE
      WRITE(KDV,'(A)')
     1      '- MAXIMUM AND MINIMUM VALUES OF AN ARRAY.',
     1      '  ------- --- ------- ------ -- -- ------',
     1      '- MAXIMUM   LOCATION  MINIMUM  LOCATION',
     1      '  -------   --------  -------  --------'
C
C*          3.2     PRINT MAXIMUM, MINIMUM, AND THEIR LOCATIONS.
C
  320 CONTINUE
      WRITE(KDV,'(1H0,F8.4,I7,4X,F8.4,I7)')
     1      ZMAX,IMAX,ZMIN,IMIN
C
C     ------------------------------------------------------------------
C
      END
```

P R E F I X	T Y P E
I, J, K, M, N	INTEGER
L, O, G	LOGICAL
C, H, Y	CHARACTER
ALL OTHER	REAL

Fig. 4.3: Variable types

The default type convention may be established in FORTRAN by the statement:

IMPLICIT LOGICAL (L,O,G), CHARACTER*8(C,H,Y).

All variables not typed according to this default convention must be declared explicitly (e.g. in REAL, INTEGER, etc. declarations).

In addition, a prefix convention is used to indicate the STATUS or SCOPE of the variable. This enables differentiation at a glance between

- COMMON or GLOBAL variables
- LOCAL variables
- dummy arguments to subroutines
- loop control variables
- PARAMETER variables.

Fig. 4.4 defines the prefix conventions for this purpose.

TYPE / STATUS OR SCOPE	INTEGER	REAL	LOGICAL	CHARACTER
GLOBAL OR COMMON	M, N	A to F Q to X	L (BUT NOT LP)	C
DUMMY ARGUMENTS	K	P (BUT NOT PP)	O	H
LOCAL VARIABLES	I	Z	G	Y (BUT NOT YP)
LOOP CONTROL	J (BUT NOT JP)	-	-	-
PARAMETER	JP	PP	LP	YP

Fig. 4.4: Prefixes indicating variables by status

4.6 Conventions for Comments

All comments within the code should take the form of titles, clauses, or sentences terminated by full stops or periods. Attention should be given to source code readability, and to the desirability of extracting documentation from the source code. Following the description given at the head of each routine, documentation in the routine body should not be over-elaborate. Too many comments within the code body make the logic of the coded statements difficult to follow. Short, to the point titles to sections and sub-sections, coupled with details of branches should read as a statement by statement algorithm outlining the major logical steps. Little more by way of comments should be necessary.

Where upper and lower case characters are supported within the source code maintenance facilities these should be used to enable good quality documentation to be extracted.

Where upper case only is supported, an extraction programme capable of conversion to mixed upper/lower case will be provided, requiring the following conventions. Abbreviations terminated by full stops (e.g. etc.) should be either avoided or coded without the full stop (eg etc). Proper names appearing in the middle of a clause or sentence should be prefixed by an asterisk (*), indicating an upper case first letter to the document extraction programme. Alternatively, proper names may be parenthesised by asterisks (eg *FORTRAN*) to indicate that the whole word is to be extracted as upper case letters. Header titles and section titles should be "underlined" by coding - under each character in the next consecutive statement (including punctuation characters). Underlined titles will be recognised by the document extraction programme and printed in upper case. A single letter followed by a full stop will be assumed to be an initial, and extracted as upper case.

With this convention, the following rules apply to an upper/lower case documentation extraction programme:

- the first letter after each full stop is UPPER CASE;
- words that are underlined are UPPER CASE;
- words bracketed by asterisks are UPPER CASE;
- words prefixed by a single asterisk have first letter UPPER CASE only;
- single letters followed by full stops are UPPER CASE;
- all other letters are LOWER CASE.

4.7 Source Code Documentation

The production of documentation is a skill at least as important as the skill required to design and generate good code. It enhances the value of the code, aiding

- understandability
- modifiability
- maintainability
- transportability
- recoding, if this should ever be required.

The documentation of a system is difficult to keep up to date. Source code documentation is easier to maintain in this respect, as it should be changed when the code itself is changed. There is thus obvious value in maintaining source code documentation that can be extracted to form the basis of external documentation. Hence the conventions for comments in 4.6 above.

Three levels of document extraction are supported:

- OVERVIEW
- EXTERNAL
- INTERNAL

In addition, source code statements of value to the documentation (e.g. critical DATA statements, special assignment statements, etc.) may be extracted.

OVERVIEW documentation is triggered by C**** in columns 1 to 5 (C and * in column one are interchangeable - either may be used). It is terminated by the first non-comment, or by triggers for EXTERNAL or INTERNAL documentation. The trigger must contain a title. It is suggested that OVERVIEW documentation comprise the title and purpose of each routine.

EXTERNAL documentation is triggered by C** in columns 1 to 3; it also includes OVERVIEW documentation. It is terminated by the first non-comment, or by a trigger for INTERNAL documentation. The trigger must contain a title. It is suggested that EXTERNAL documentation comprise, in addition to the OVERVIEW:

- the INTERFACE details
- the EXTERNALS used

- the METHOD used
- a REFERENCE to further documentation
- the AUTHOR and MODIFICATION details.

INTERNAL documentation is triggered by C* in columns 1 and 2; it also includes OVERVIEW and EXTERNAL documentation. It is terminated by a non-comment. The trigger must contain a title or comment. It is suggested that INTERNAL documentation comprise, in addition to OVERVIEW and EXTERNAL:

- details of COMMON block variables
- section titles
- sub-section titles
- details of branches (referring to section and sub-section branched to, not to label numbers)
- any appropriate additional comments.

The extra information produced when INTERNAL documentation is extracted should represent, in statement form, the algorithm used.

Additionally, any statements enclosed by comments containing C*** in columns 1 to 4 will be listed with EXTERNAL or INTERNAL documentation.

Fig. 4.2 illustrates the conventions for source code documentation. Figs. 4.5 to 4.7 contain examples of the 3 levels of documentation after extraction.

```
SUBROUTINE EXAMPLE(PA,KLEN,KDV)                                  86/05/15

   EXAMPLE  - routine to demonstrate the  DOCTOR  style.

   PURPOSE.
   --------

      The main purpose of this routine is to demonstrate the
   conventions, style, and presentation of source code using a
   coding standard based on the  DOCTOR  system.

      Additionally, this routine prints the maximum and minimum
   values of an array.
```

Fig. 4.5: Example of Overview Documentation

86/05/15

SUBROUTINE EXAMPLE(PA,KLEN,KDV)

EXAMPLE - routine to demonstrate the DOCTOR style.

PURPOSE.

The main purpose of this routine is to demonstrate the conventions, style, and presentation of source code using a coding standard based on the DOCTOR system.

Additionally, this routine prints the maximum and minimum values of an array.

INTERFACE.

CALL EXAMPLE(PA,KLEN,KDV)

PA	-	array to be examined (input)
KLEN	-	length of PA.
KDV	-	logical unit for output messages.

METHOD.

PA is scanned, and the minimum and maximum values extracted. These are then written to file KDV.

EXTERNALS.

None.

REFERENCE.

None.

AUTHOR.

J. K. Gibson ECMWF 14/05/86.

MODIFICATIONS.

None.

Fig. 4.6: Example of External Documentation

```
SUBROUTINE EXAMPLE(PA,KLEN,KDV)

 EXAMPLE  - routine to demonstrate the  DOCTOR  style.

PURPOSE.
--------

     The main purpose of this routine is to demonstrate the
conventions, style, and presentation of source code using a
coding standard based on the  DOCTOR  system.

     Additionally, this routine prints the maximum and minimum
values of an array.

INTERFACE.
----------

      CALL   EXAMPLE(PA,KLEN,KDV)

           PA      -  array to be examined (input)
           KLEN    -  length of  PA.
           KDV     -  logical unit for output messages.

METHOD.
-------

      PA  is scanned, and the minimum and maximum values
extracted. These are then written to file  KDV.

EXTERNALS.
----------

     None.

REFERENCE.
----------

     None.

AUTHOR.
-------

     J. K. Gibson       ECMWF       14/05/86.

MODIFICATIONS.
--------------

     None.

     1.      SET INITIAL VALUES.
             --- ------- ------

     2.      EXTRACT MAXIMUM AND MINIMUM.
             ------- ------- --- -------

     3.      PRINT RESULTS.
             ----- -------

     3.1      Print heading.
     3.2      Print maximum, minimum, and their locations.
```

Fig. 4.7: Example of Internal Documentation

5. A MAINTENANCE STANDARD

5.1 Avoiding the Maintenance Problem

The simplest and most effective way of avoiding the maintenance problem is to live with the software, make no changes, and do no maintenance! As this is clearly impracticable and undesirable the problem is really to minimise the amount of maintenance, and to maximise the ease with which such maintenance as should prove necessary can be achieved.

As has been indicated above, the main means of minimising maintenance effort is to produce well designed, well documented and well written source code. If future modifications are planned for, their implementation will be facilitated. If software transportability is planned for, software migration will be facilitated. If code is easy to understand, it will be easy to modify. Thus the design standard and the coding standard are essential pre-requisites to the maintenance standard.

Additionally, there is a need to produce maintenance documentation, to plan changes, and to keep a software log.

5.2 Maintenance Documentation

The primary maintenance documentation is the source code. Only the source code can be guaranteed to be "up to date". All secondary documentation, though useful, cannot be guaranteed to be as up to date as the source code.

It is thus of vital importance to update source code comments when updating the source code.

The document extraction facility used to extract "internal" documentation should provide the narrative section of a system reference manual. This should be expanded as necessary, and should contain a section comprising a system maintenance log. The maintenance log should contain a diary of modifications, enhancements and changes introduced, together with a record of problems encountered, and subsequent remedial action.

Maintenance documentation, as described above, should provide useful guidance in the planning of changes, and in the correction of errors. It should be kept as up to date as possible.

5.3 Modifications and Enhancements

There are 3 levels of modification for the purpose of the application of this standard:

- minor changes
- replacement/addition of routines
- system re-write or addition.

Where minor changes are required to an existing system, these should be carried out with a minimum of disturbance to the existing system. Documentation should be amended where necessary, and the maintenance log updated. If a number of changes are required to any routine not conforming to the coding standards in 4. above, consideration should be given to replacing the entire routine.

Where it is necessary to replace or add one or more routines, the coding standard in section 4. and this maintenance standard should be followed. Additionally, appropriate items such as planning, design, software review, and testing procedures should follow the recommendations within the software development standard in section 3. If a significantly large number of routines within a system need to be replaced, consideration should be given to the development of a replacement system.

Where it is necessary to rewrite or add a new system, the software development standard (section 3), the coding standard (section 4) and this maintenance standard should be followed.

6. ACKNOWLEDGEMENTS

The author wishes to acknowledge, with gratitude, invaluable material presented by James Martin and Carma McClure in "Software Maintenance: The Problem and its Solutions" (Prentice-Hall, 1983). The many useful references in this book provide a wealth of additional material. Acknowledgement is also due to Professor K.V. Roberts, for had it not been for the OLYMPUS system I doubt whether programming support for ECMWF's forecast models would have reached the standards of today.

REFERENCES

Frank, R.H., 1971, DOCK - an INTERNAL/EXTERNAL documentation processor (Copyright Control Data Corp.). (Extracted from the source code by courtesy of Frank Stevens, CDC).

Gibson, J.K., 1980, Programming Systems, Documentation, OLYMPUS, DOCTOR. ECMWF Technical Memorandum No. 20.

Roberts, I.V., 1974, The Olympus Programming System Computer Physics Communication, 7, 237-240.

Gibson, J.K., 1982, The Doctor System - A DOCumenTary ORiented Programming System, ECMWF Technical Memorandum No. 52.

Dijkstra, E., 1976, Structured Programming, Software Engineering Techniques, Petrocelli/Charter.

Martin, J., and McClure, C., 1983, Software Maintenance: The Problem and its Solutions (Prentice-Hall).