DATE: March 5 1981

TO: RD&E

FROM: Bridget Hale and John Holloway

SUBJECT: Structured Programming System

REFERENCE: None

# SPS - Structured Programming System

## Abstract

In December 1979 a project was initiated in Bedford to review the Structured Design and Programming Techniques in use both in R&D (UK) and outside Prime.

The interest in such techniques suggested that the project should go on to create:

a simple, but logically consistent, set of tools
and
a short Programmer's Guide

This document describes the results and is, in effect, the Programmer's Guide.

# Table of Contents

## Changes from Previous Version

All tools have been extended to process files with the following suffices:

.LISP
.EMACS

and the previously supported suffix .BASICV has been changed to .BASIC to bring it in line with the file naming standards.

A new tool has been added - TEMPLATE - which builds a dummy SPS module (including Prime Copyright lines).

Some changes have been made to DENOTE with respect to the manner in which a Book List is processed.

# 1 Introduction

Today there is a great deal of interest, within the Computer Industry, in something often rather abstractly described as 'Structured Programming'.
Much of this is encouraged by the fact that rarely a week passes without the publication of an article in the trade press, on some new technique or variant of an existing technique; or the publishing of yet another book on that same subject.

Opinions on the benefit of the use of these techniques vary greatly, from:

> The belief that a 'junior' member of staff using a Structured Programming technique is as good as, or better than, the most experienced member of staff (and is certainly cheaper).

to:

> The belief that the whole theory of Structured Programming can be summarised in the phrase 'Ban the GOTO'.

In answer: to the first of the above, no Structured Programming technique can replace what a person lacks in either experiance or ability, in fact a misguided or inappropriate use of such techniques may reduce the performance of a 'good' and/or 'experienced' programmer; to the second, that it is a naive misjudgement of the scope of Structured Programming.

In many ways 'Structured Programming' suffers from the word 'Programming' in its name in that it is frequently dismissed as only being applicable to programmers.

In actual fact the term Structured Programming covers a wide range of ideas/techniques ranging over all of the following:

- programming libraries
- structured coding
- design methods
  - e.g. Top-Down
    - Data Driven
    - etc.
- project administration
- project organisation
- documentation
- standards

and many more.

Unfortunately where Structured Programming techniques are used this is frequently in a rather random fashion and thus the major benefits to be gained from such an approach are invariably lost.

Where Structured Programming is used well the following should arise:

1. A general awareness of different design techniques and their relevance to particular stages of the design process.

This serves to strengthen a design and results in methodologies being established over a period of time.

2. The combination of a set of techniques that complement one another thus re-inforcing the benefits to be gained from such an approach. It is important to appreciate that the random use of disjoint techniques (in the worst case) can actually damage a project.

3. The reduction of the cost of a product when considered over the whole of a product's life-cycle. At the current time the only costs predicted when estimating the cost of a product are those that are incurred during development.
In many cases these costs are just the tip of an iceberg. The real costs only become visible years later when the costs of the following can be calculated:

   a) customer support
   b) 'bug' administration
   c) quality assurance (repeated for each update)
   d) loss of goodwill and/or upgrade orders from unhappy customers
   e) bug fixers
   f) delays to new projects as a result of re-assignment of personnel for 'short' periods away from current assignments to maintenance

These costs only disappear totally when a product is withdrawn or re-implemented.

The SPS project has attempted to look at what is already in use in the department and at what people would like to see. This has been done with the aim of optimising the above.

## 2 SPS Project

The SPS project was started by holding sessions with as many of the members of the organisation as was possible, requesting input on the following topics:

design techniques
design languages
program layout
documentation
structured coding methods
tools

People were thus given the opportunity to express their views on what they did, what they would like to do and what they would like to help them.
The opinions expressed by our collegues at that stage of the project ranged over a surprisingly large area and are collated and summarised in the document U-0024, these opinions are also included here as an appendix.

The project then went on to try to link this input to the more popular/prominent ideas from outside, to try to satisfy people's requirements in conjunction with maximising the benefits to be gained from the adoption of Structured Programming techniques.

This document describes the results produced by the SPS project. It describes the tools produced by the personnel assigned to the project and can act as a programmer's guide to the use of these tools. This document also draws attention to a number of areas where further work could be done to either establish standards or to produce further tools.
It is hoped that in the future as projects establish the need for a tool, that they will develop these in a manner such that they will be consistent with the SPS package and will be donated to SPS. In this manner the SPS project will continue to live and grow.

The use of the SPS tools and style by a project should lead to a consistent and systematic approach to project organisation and product development.
This should over a period of time lead to an increase in the quality of the products produced, reduce the effort required to achieve this quality, and a visible decrease in the amount of maintenance required by such products as a consequence.

When reading the sections within this document it is important to realise that the requirements/behaviour of the various tools and techniques are highly dependant on one another. For this reason the requirements of a particular language environment when applied to the items described in this document are discussed in a series of appendices.

# 3 Routine Format

## 3.1 Requirement

From the beginning of a project information is being built up about the project's requirements, this then evolves and grows into a design from which a program is eventually coded.

This information has always existed in some form for a project but has not always been maintained on the machine. In view of the distributed locations of R&D and the presence of software for transmitting information from one machine to another there is a very strong reason for putting this information onto the machine.

However, it is also important that such information be trusted, by those who refer to it, to be an up-to-date representation of that information. In order that the ease of updating such information be increased it is proposed that information associated with a particular routine be located with the source of that routine.

A format for a routine is therfore proposed as follows:

```
TITLE: the identity of the routine

START-DESCRIPTION:
     This is a block of narrative describing the function of the
     routine.
     At a project level a decision may be taken as to the type of
     information included here.
END-DESCRIPTION

START-DESIGN:
     This is a block of design information.
     The design may be expressed in any form suitable for
     inclusion in a text-file.
END-DESIGN

START-CODE:
     The program
END-CODE
```

In addition to the above the following may be present at the start of the file in which the routine(s) are held:

```
the mandatory first 3 lines (see PE-A-43)

START-HISTORY:
     This is a block of history information created, either
     manually or automatically, (when either the Source File
     System or a callable editor in conjunction with CPL is used)
     to control the generation and modification of the file.
END-HISTORY
```

The routine format recommended will encourage people to maintain project information of both a descriptive and a design nature within their programs.
The descriptive information should be created initially and modified, as appropriate, when the routine is implemented.
The description should not be too detailed when originally written; in this way it will survive many detailed changes in the succeeding coding without the need for alteration.
Once the first implementation has been produced, it is impossible to make any logical changes to a routine without consulting the narrative and the design. With the suggested format these will be at hand. It only makes sense to change them at the same time as changing the code.

The information contained in these blocks contributes substantially to the ease with which a project may be maintained and also to the production of a project's internal documentation.

The format shown above only indicates a single occurrence of narrative, design and code. In practice a routine may have any number of these blocks in any sequence.

## 3.2 Title

The function of the title line is to identify the routine. This may appear to be cosmetic, but it does enable the user to identify individual routines when multiple routines are included in a file. It also identifies routines to some of the tools described later in this document.

The format of the title line is:

TITLE: <name>

where <name> is any sequence of characters.

This format is affected by the commenting requirement of the language in which the routine is written. This is described fully in a series of appendices.

## 3.3 Description

The function of the Description Block is to describe the function of the routine in as detailed a level as is considered to be appropriate, in a location where it can be easily found.
It may also be the case that some or all of the following should be included:

parameter definition/description
externals definition/description
abnormal conditions definition/description

It is a responsibility of a Project Leader to define what should be present in a Description Block for a particular project. It should however be remembered that one of the subsequent uses of this block will be by a maintenance programmer, who may or may not be familiar with the routine.

The format of the Description Block is:

```
START-DESCRIPTION:  [<name>]
     block of narrative
END-DESCRIPTION
```

where <name> is an optional sequence of characters that may be included for information purposes. (If present this will be processed by some of the tools described in this document.)

This format is affected by the commenting requirements of the language in which the routine is written. This is described fully in a series of appendices.

## 3.4 Design

The function of the Design Block is to document the design of the routine in as detailed a level as is considered to be appropriate, in a location where it can be easily found.

No comments will be made at this point about what design technique should be used here, other than that it must be capable of being expressed in this block.

The design information can be expressed in many forms, for example:

- pseudo-language
- decision tables
- flow charts
- SADT charts
- Warnier-Orr diagrams
- etc.

It may, however, be the case that some design techniques are harder to express in a Design Block than others. What is to be stressed at this point is that the need for this information to be documented is of paramount importance.

The format of the Design Block is:

```
START-DESIGN:  [<name>]
     block of design information
END-DESIGN
```

where <name> is an optional sequence of characters that may be included for information purposes. (If present this will be processed by some of the tools described in this document.)

This format is affected by the commenting requirements of the language in which the routine is written. This is described fully in a series of appendices.

## 3.5 Code

The function of the code block is to identify the presence of the code.

It is of course obvious that most compilers can recognise code. The presence of this block may conceivably be useful to some future tool that performs some, as yet unknown, processing on a routine.

The format of the Code Block is:

```
START-CODE:  [<name>]
     Source code of routine
END-CODE
```

where <name> is an optional sequence of characters that may be included for information purposes.

This format is affected by the commenting requirements of the language in which the routine is written. This is described fully in a series of appendices.

It would of course be possible to make unlimited pronouncements on how people should code their programs. This is not within the scope of the SPS project.

A number of general points, however, can be made:

1. Structured coding principles should be adhered to, though not to the exclusion of common sense.

2. The code should be indented.

3. Any naming conventions should be documented, either at a routine level or at a project level, as appropriate.

4. 'Block' comments should be used rather than 'end-of-line' comments whenever possible, particularly in the case of block structured languages. Any formatting performed by a tool may disfigure end-of-line comments and decrease their usefulness.

5. Guidelines could be provided to indicate a mapping between the constructs of commonly used design techniques and those available in the more commonly used implementation languages.

and obviously many more.

## 3.6 The Lifecycle of the Routine

A lot has been written by various people on the subject of 'Top-Down Design' and 'Step-Wise Refinement'. In actual fact, a majority of people design their programs using this sort of approach, whether consciously or unconsciously.

As one thinks through a problem one naturally decomposes it into its constituent parts, each of which may be further decomposed. It is this approach that ensures that any problem unit is not so large that it cannot be comprehended.

Any module is an elaboration of the parent module that spawned it, and is itself the controlling parent for its own child routines.

It is proposed that a routine will start off its life as little more than a title and a block of very high level narrative. As time goes on the level of this narrative will become more detailed. This narrative should not be viewed as something that can be disposed of at some point along the development path but as something that lives and grows with the project. It will later be extracted to become part of an external design description.

This can now be used as input to a Walkthrough at which the conceptualisation of the product is reviewed and validated, and its interface to its operating environment is verified. If a true 'Top-Down' approach is adopted then the Description Blocks of all routines will exist before progressing to the next stage.

The next stage in the life of a routine is to take the content of the Description Block and express this as a Design Block. At this stage more detail becomes apparent in the expression of the routine. This design information is also alive and able to grow.

Just as the Description Block could be used as input to a Walkthrough so also can the Design Block.

It is only when the design is known and validated that the work of transcribing the design into the corresponding code should take place.

It is generally accepted that the easiest errors to locate and correct are those that are introduced at the programming phase. The earlier that an error is introduced into the expression of a problem, the wider are its repercusions, and the harder and more expensive it is to correct.

The life cycle described above effectively asks the implementor to express his problem 3 times:

as narrative
as design
as code

This provides for up to 3 levels of Walkthroughs and should minimise

the errors that are located at project integration time.

# 4 Design Expression

## 4.1 General

A large number of Design Techniques/Languages are now available for use, and many of these are used to some extent within the department.

Information is already available within the R&D (UK) Library on a number of techniques and there are a number of additional books already on order. At an introductory level the notes from a course entitled:
'Software Engineering - The Key to Quality Systems'
provide an overview of many of the techniques currently available.

A design may be expressed in any number of forms, for example:

pseudo language
decision tables/trees
block charts
finite state diagrams
SADT charts
Warnier-Orr diagrams
Data Structure diagrams
etc.

At a general level, no one of these techniques can be said to be 'best' for all design requirements. Usually a combination of a small number of these techniques will be appropriate in solving a single problem. Different techniques may of course be better suited to different problems.

The importance of ALL of these formal techniques is that they force the designer to express a design in a formal fashion, as oposed to just diving in to the implementation phase.

It is important that any design technique used should be documented for the benefit of anyone not familiar with that technique.

When selecting a design technique, the following should be considered:

1. The environment in which it is to be used:
   a) A technique suited to the analysis stage may not be well suited to program design.
   b) There does not need to be a correspondance between the constructs provided by a design technique and those available in the eventual implementation language.
   c) The constructs provided by a design technique should be (or be used) at a higher level than those available in the implementation language. Most people have seen flowcharts in which a box contains the statement 'a=10', this is not a fault of the flowcharting technique but of the person using it.
   d) The technique should approach natural language as much as

possible (within the constraints imposed by the problem).

e) The representation chosen to express the design should be such that the transformation to code is straightforward, and not error-prone.

2. The mechanism used to express a design:
   a) The process of creating a design must be simple.
   b) It must be possible (and easy) to comprehend (and maintain) the expression of a design.
   c) It must be possible to circulate design information to interested parties (even if these are spread over a number of distributed sites).
   d) Evolution of a design expression must be possible.

3. The number of design techniques used within a project should (if possible) be kept to a minimum.

If any long term benefits are to be gained from the systematic use of any design technique then it is important that people be encouraged to maintain and evolve the original design expressed using that design technique.

If the design is maintained to reflect any changes made to a program then the design can take its rightful place in a project's documentation.
It should never be necessary to re-create the design of a product after it has been implemented for the purpose of producing documentation.

If the program is located physically alongside the design then the probability of their being in-line with one another is increased and this fact can be used to help produce the much needed documentation.
This can be done most easily if the design technique used is text based rather than diagram based.

## 4.2 STROMA

STROMA is a dialect of pseudo language that has been produced by the SPS project.

After discussions with many of the people in the department, it appeared that many people are using either a pseudo language dialect, or simply expressing their design in English, by way of Structured Comments.

The most commonly used dialect of pseudo language used within the department is known as R-Notation.
Unfortunately a number of personal mutations have been introduced into this dialect now that it is being used with PL/I as an eventual implementation language rather than Assembler.
In some extreme cases the correspondance between a PL/I program and its R-Notation design is of a one-to-one nature.

When English is used to express a design, ambiguity and impreciseness

can be introduced due to the manner in which people tend to express themselves. Also any English description tends to contain only sequential information.

The intention in designing STROMA was to encourage people to express themselves in something akin to Structured English.
For this reason a number of structuring constructs have been defined, but no rules have been created as to what should be written within any of these constructs.
However, as with any design technique its success or failure as a technique depends on the user.

Many learned persons, such as Edsger Dijkstra, have expressed the opinion that all programs can be built from a combination of elements known as Sequence, Selection and Iteration.
At a design level the requirement is rather that the constructs available to the designer can be decomposed in a predictable manner into these 3 elements as appropriate to the eventual implementation language.

STROMA will be defined in the following sub-sections.
In the examples that appear in this section STROMA structuring words are capitalised and constructs are formatted to emphasise their control structure.

An example of its use is:

```
denote:
   BEGIN
      DO initialisation
      REPEAT UNTIL no more input files
         DO file processing
      END-REPEAT
      DO termination
   END

initialisation:
   BEGIN
      set up parameter defaults
      analyse parameters
      SELECT
      WHEN output file exists
         allow choice of alternative file
      ELSE output file does not exist
         NULL
      END-SELECT
   END

file processing:
   BEGIN
      open input file
      output file information line with RUNOFF control
      read a line
      REPEAT WHILE not end of file
```

```
            SELECT
            WHEN title line
               output title information line with RUNOFF control
            WHEN start line of description or design
               indicate that lines are to be output
            WHEN end line of description or design
               indicate that output to cease
            ELSE ordinary line
               output line if required
               a hook table search could be done at this point
            END-SELECT
            read a line
         END-REPEAT
         close input file
      END

   termination:
      BEGIN
         close files
      END
```

### 4.2.1 Labelling

Any construct may be labelled.
A label is any sequence of characters followed by a ':' and should appear on a line on its own.

e.g.
```
  get next item:
```

The purpose of the label is to allow identification of a portion of the design.
This identification is purely for identification purposes, except for the case of labelling a portion of the design that has been invoked.

Programming Considerations:
For any label appearing in the design there should be a corresponding label appearing in the code (within the limitations of the programming language being used).

### 4.2.2 Invocation

At any point in the code it is possible to invoke a design unit that appears elsewhere.
An invocation consists of the word 'DO' followed by a name.

e.g.
```
  DO get next item
```

A name that is invoked should correspond to a label appearing in the design unless it is the name of an external module.
It may be convenient to indicate that a design unit is defined

externally by including the word 'EXTERNAL'.

e.g.
```
DO EXTERNAL tnou
```

If additional information is to be supplied about the invocation then this may follow the name; a ':', '(' or '[' may be used to separate the name from such comments.

Programming Considerations:
The use of the invocation construct in the design does not necessarily imply that a subroutine call will be implemented.

### 4.2.3 Invoked Design Units

A section of the design that is invoked consists of a mandatory label, followed by 'BEGIN', followed by a portion of design, followed by 'END'.

e.g.
```
get next item:
   BEGIN
      comment sequence
   END
```

The word 'END' implicitly causes a return from the invoked unit to the construct following the invocation.

### 4.2.4 Sequence

A sequence of comments may appear at any level in the design, and consists of one or more comments.
Each comment should be written on a new line.

e.g.
```
set up parameter defaults
```

Programming Considerations:
Design language statements should not normally correspond to a single programming language statement.

### 4.2.5 Selection

The 'SELECT' construct provides the designer with a mechanism for defining multiple choices. This construct has the following format:

```
SELECT
WHEN condition definition
   comment sequence
WHEN condition definition
   comment sequence
```

```
.....................
ELSE condition definition
   comment sequence
END-SELECT
```

The 'ELSE' part of this statement is mandatory.
Its function is to ensure that the designer has given some thought to the question of what happens to the conditions that are often not specified.
It may be the case that in practice the 'ELSE' part of the construct often contains only an instruction to do nothing. (A special word is introduced later for expressing this.)
Only one of the multiple choices is ever selected.
A condition definition may involve one or more conditions.

e.g.

```
SELECT
WHEN temperature>20
   DO warm processing
WHEN temperature<5
   DO cold processing
ELSE 5 <= temperature < 20
   no action required
END-SELECT
```

Programming Considerations:
If a selection requires to be made between more than two choices then it should be expressed as such in the design, even though the implementation language may restrict the implementor to a two way choice.

### 4.2.6 Iteration

The 'REPEAT' construct provides the designer with a mechanism for defining the control of a loop. This construct has the following format:

```
REPEAT repetition definition
   comment sequence
END-REPEAT
```

The repetition definition must be one of:

```
WHILE condition definition
UNTIL condition definition
FOR control description
```

When the 'WHILE' option is used, the test involved is performed at the start of the iteration. It is therefore possible that no iterations may result from this form of the construct.
When the 'UNTIL' option is used, the test involved is performed at the end of the iteration. It is therefore the case that at least one iteration will always occur.

When the 'FOR' option is used, the wording of the 'control description' should imply how the control is to be implemented. It is possible that no iterations may result from this form of the construct.

A repetition definition may involve one or more of the definition clauses.

e.g.

```
REPEAT WHILE not end of file
   DO file processing
END-REPEAT

REPEAT UNTIL end of file
   DO file processing
END-REPEAT

REPEAT FOR each person on payroll
   DO produce payslip
END-REPEAT
```

### 4.2.7 Event Handling

The 'MONITOR' construct provides the designer with a mechanism to define responses to events. These events may be either 'hard' or 'soft' events; 'hard' events being automatically detected, whereas 'soft' events have to be detected explicitly.
A single 'MONITOR' construct may be concerned with more than one event. This construct has the following format:

```
MONITOR condition list
   comment sequence
   if none of the conditions occur then
   control passes to the end of the statement
ADMIT condition list
   actions corresponding to these conditions
ADMIT condition list
   actions corresponding to these conditions
.....................
END-MONITOR
```

Note that each condition being monitored must have a corresponding 'ADMIT' clause.

When a 'soft' condition is being explicitly detected a special construct 'BREAK' is used in conjunction with that condition name.

e.g.

```
MONITOR break-key
   process file
ADMIT break-key
   DO controlled shut-down
   exit from program
END-MONITOR
```

```
REPEAT UNTIL end of file
   MONITOR end of tape
      read record from tape
      SELECT
      WHEN end of tape
         BREAK end of tape
      ELSE not end of tape
         process record
      END-SELECT
   ADMIT end of tape
      BEGIN
         DO get next tape
      END
   END-MONITOR
END-REPEAT
```

Programming Considerations:
Extreme care should be taken when implementing a 'MONITOR' construct. Each 'soft' event could require two 'GOTO' statements and each 'hard' event could require one 'GOTO' statement.

### 4.2.8 Special Words

A number of special words have been defined to enable a number of standard requirements to be expressed in uniform ways. These can each occur as a comment.

'BREAK' normally causes a 'return' up one level of control.
Its other use is in the 'MONITOR' construct when used in conjunction with the detection of a 'soft' event, when it causes control to be passed to the corresponding 'ADMIT' clause.

'NEXT' is only meaningful in a 'REPEAT' construct. It causes the next iteration to be commenced.

'NULL' is used to indicate that no comment is present. This will typically be used in the 'ELSE' part of the 'SELECT' construct.

## 5 Project Libraries

All project work items are currently organised into project ufds for the purpose of handing a product over to SDI.
At this time there are few requirements as regards to the organisation of such a ufd, other than that the command files supplied with the product should take account of any internal organisation.
Different projects adopt different organisational arrangements for their project work items for this handover.
The format of the ufd organisation during development does not always match this structure. Frequently people favour the idea of working in their own ufds and only move software into a project ufd for integration and final testing.

It is, however, very important that all members of a project team and any other interested parties be able to locate any project work item with ease.
This can best be done if a project ufd and its associated structure is defined for any project.
The 'project library' reflects the fact that a team of people work together with the common objective of creating one product. In the past one often saw several individuals going about their work in an individual way and only coming together on the great day of 'integration'. It is not surprising that sometimes their separate components did not match one another.
Nowadays we see the development process as more co-operative, people will discuss and negotiate interfaces or functions, there is a sense of collective responsibility for the whole product. The project ufd is established to reinforce the team and answer their day to day need for information.
With a project 'library' it is no longer necessary to rely on fallible memory, hurredly written notes or second source rumours to understand an interface or function. If the facts have been defined at all, they will be easily found within the 'library'.

Some of the benefits to be gained from creating a project ufd in which all work items are created/kept are as follows:

1. The total assets of a project are kept together and are available to any interested party.

2. Appropriate backup and recovery procedures can be adopted for a project as a whole.

3. Standard procedures can be created for product building that are based on the organisation adopted.

4. A working space is set aside for a project and resources can be more easily be assigned to a project.

5. File naming conventions can be established for a project and the conformance to such standards is easily visible.

6. Any questions about a project should be resolvable by reference to

the project ufd since all work items reside there.

7. The project ufd represents a source of status information for all project members and for management.

8. The project ufd provides an opportunity to resolve contention for central project resources such as subroutine libraries.

No structure for a project ufd can be laid down arbitrarily. It can only be said that whenever possible the project leader should create a ufd structure or structures that permit project members to easily fulfill their project responsibilities.

In deciding on a ufd structure the existence of the following should be considered:

multiple product units within a large project
documentation
source files
binary files
command files
testing requirements
subroutine libraries

together with any other considerations specific to a particular project.

## .1 Project Catalogue

The Project Catalogue is a definitive list of the resources of a project.
By maintaining a list, in a known place, of all source files, insert files, and any other interesting material, that project clearly identifies its resources.

Such a list could be used in conjunction with Project Specific CPL utilities to perform project specific tasks regardless of the structure of the Project Library.

For example, a CPL utility could be written to compile each file listed in the Project Catalogue using the compiler appropriate to the file's suffix. Such a utility could recognise that certain entries in the catalogue (e.g. insert or design files) are not eligible for compilation.

An entry in the catalogue must consist of the full treename of the file and any parameter options required/recognised by project utilities.

A use of the Project Catalogue is described for the tool DENOTE (later in this document).

## 6 Walkthroughs

The Walkthrough was mentioned briefly in the section on the Lifecycle of a Routine.
It was said there that at a Walkthrough

'the conceptualisation of the product is reviewed and validated'

and that the lifecycle described in that section effectively asks the implementor to express his problem three times:

as narrative
as design
as code

thus providing for up to three levels of Walkthrough.

In actual fact those three instances of Walkthroughs in a Routine's Lifecycle are but a few of the possible instances when a Walkthrough can be usefully held during a Project's Lifecycle.

In any project anyone with a particular problem tends to discuss a problem area at great length with either the Project Leader and/or a collegue. The process of explaining a problem can often make a solution visible, and/or the other person may see a solution or provide useful ideas.
However, it is frequently the case that large areas of a product are not seen by anyone other than their originators simply because there appear to be NO problems in these areas.

In the situation where Walkthroughs are employed then the whole project will be examined. Obviously some areas will still be considered to be straightforward and therefore receive less attention than known 'problem' areas, but the whole product will be reviewed.
As a result of this approach a situation should develop where the project team are satisfied as regards to the correctness of the design (and the code), the accuracy and style of the implementation, the exhaustiveness of testing, and last but by no means least the quality of the documentation.

A number of things can be achieved as the result of a Waklthrough; these include some or all of the following:

1. The introduction of a work item to the project team by its originator - after a successful Walkthrough the project effectively takes collective responsibility for that work item.

2. The review of the development of an existing work item.

3. To catch any errors (in code and/or design) as early as possible in a project with a view to minimising their effect and the subsequent cost of correcting them.
   Typical of the kind of errors that can be detected and/or prevented are:
   - those arising out of interface problems/incompatibilities
   - missing functions
   - misinterpreted functions

4. To involve project members in as much of the project as possible and to increase their awareness of the project's state at any time.

5. To instill in project members a 'total' responsibility for a 'whole' product.

6. To monitor progress.

7. To obtain advice from any 'experts' invited to the Walkthrough.

8. To monitor the project implementation style.

9. To verify that documentation reflects the state of the project.

10. To facilitate the exchange of information through the project members.

If Walkthroughs are to be used then they should be instituted as early in the project's lifecycle as possible and be scheduled to take place at a number of specific points in that lifecycle.

A Walkthrough should be held at the beginning of a project to discuss the Marketing Requirements and/or Base Document of the project and ensure that all project members understand what is required to be produced.

The next step is usually to produce a Functional Specification. This crucial document can be reviewed section by section in Walkthrough style discussions. When that specification is agreed, the design work can begin in earnest.

If the proposed lifecycle for a routine is being followed, the first job is to write Description Blocks for the main routines.
These may be keyed in and printed via DENOTE or just left hand written, since, at this stage, details are still being tied down.
A Walkthrough can then be held on the Description Blocks established for the planned routines. This makes sure that the embryo routines are being conceived along the right lines before detailed design is committed.

The next level of Walkthrough can be held on the Design Blocks corresponding to routines. At this time it will be possible to see the level of complexity included in each routine and recognise the shape of the proposed product.
The information available at this stage can lead to a re-evaluation of projected timescales. It should also allow a judgement to be made as to which routines will receive further Walkthroughs when they have been coded.

Additional Walkthroughs may be held on the code of various routines. Often routines will be selected on the basis of the complexity of their design. There is, however, no harm in also selecting a number of routines at random for examination at a Walkthrough.

If the earlier Walkthroughs served their purpose correctly few problems should be detected at this stage.

In addition to the above Walkthroughs held during the evolution of the routines that will eventually make up a product, Walkthroughs can also be held during the testing phase of development. The expression 'testing phase' is used here to encompass all activities associated with testing; thus including planning, test generation and result prediction/checking.
It may well be the case that in spite of the Walkthroughs held so far, different project members may have different ideas of what results are expected from a particular test set, and these can then be resolved.

Walkthroughs can also be held to guide the production of documentation, to review its quality and accuracy. The Functional Specification should have been examined early on in the Project's Lifecycle. The Description and Design Blocks for routines should also have been validated and combined into a Design Specification or Design Notebook, using DENOTE or some similar tool. A Walkthrough can now be held to consider any documentation produced for publication.

It can thus be seen that the Walkthrough can play a significant part in a project's development cycle when it's use is encouraged.
It is also important that the use of this tool (for it is a tool, like any of the others described in this document) be planned and scheduled into a Project Plan. If too few are held, then any (or perhaps all) benefits may be lost; if too many are held, then the project could become one long meeting and little work will be accomplished.
The correct balance between Walkthroughs and work is important and can only be judged by experiance.

If Walkthroughs are used then the following must be remembered if they are to stand a chance of being successful:

1. The work items of each project member must be seen to be subject to the same reviewing process - no one should be exempt.

2. The Walkthrough is not used when things go wrong in order to find a scape-goat for any project slippage.

3. Project members accept the usefulness of these sessions and contribute to them.

## 7 TEMPLATE

TEMPLATE is a utility which builds the outline framework of the standard SPS file construction, in a format appropriate to the requested file type.

### 7.1 Function

TEMPLATE is intended to create a shell in which the final file can be built. To this end the following will be included in the framework:

1. A Copyright Block which includes the file name, location of file, author, function and date.

2. A Title Line giving the routine name and function.

3. A History Block with the first entry being information on the date when the template was constructed.

4. A Description Block.

5. A Design Block.

6. A Code Block, provided that the file language type is not design or that the penultimate component of the file name is not .INS.

Use of TEMPLATE allows the programmer to set up 'stub' routines easily when using a 'top-down' approach to development. It also ensures that a module conforms to both PRIME and SPS file formats.

### 7.2 User Interface

TEMPLATE is invoked by:

```
TEMPLATE base[.<suffix>][control arguments]
```

where base.<suffix> is the name of the file that is to be created. If the name of the file specified is 'base' then '-<suffix>' must be specified as a control argument. If the name of the file specified is 'base.<suffix>' then '-<suffix>' must not be specified as a control argument.

The control arguments may be chosen from the following in any order.

| | |
|---|---|
| -PATH <pathname> | If specified this must be followed by the pathname of the ufd in which the file is to be created. Default pathname is the current attached UFD. |
| -NO_QUERY, -NQ | If specified this will result in the named file overwriting any file of that |

| | |
|---|---|
| | name in the specified UFD, without verification request. |
| -<suffix> | If omitted .<suffix> must be specified. A null suffix will not be accepted. This parameter may be one of the strings: 'PL1' 'PLP' 'PL1G' 'FTN' 'F77' 'PMA' 'COBOL' 'PASCAL' 'DES' 'BASIC' 'CPL' 'LISP' 'EMACS' and indicates that the file to be produced should be of the corresponding type. |

TEMPLATE will then ask for additional information with the following prompts:

| | |
|---|---|
| FUNCTION: | Mandatory. A one line description of the function of the routine. |
| AUTHOR: | Mandatory. |
| DESCRIPTION PROFILE: | An option which allows you to insert the contents of a specific file into your Description Block. (If no file to be included type <return>). This can be used to include project specific information within all project modules. |
| CODE PROFILE: | This prompt will not appear if a design file or an insert file is being built. This is an option which allows you to insert the contents of a specific file into your code block. (If no file is to be included type <return>.) |

## 7.3 Processing

TEMPLATE creates a file which contains the following:

1. A Copyright Block conforming to PRIME standards.
2. A Title Line consisting of the routine name and function.
3. A History Block with an entry showing the date the template was constructed.
4. A Description Block. If a description profile has been provided then the contents of this file will be inserted into this part of the file. If no description profile is provided then a line saying 'description to be inserted ' is inserted here.
5. A Design Block, which provides the basis for a STROMA based

design.

6. A Code Block, this block will not appear in a design file or an insert file. If a code profile has been provided then the contents of this file will be inserted into this part of Template. If the language type is PL/I, this block will additionally contain, a label, a dummy Procedure declaration and an end.

The following table links the format of the contents of a file to its corresponding suffix:

| contents | suffix |
|---|---|
| PL/I | .PL1 |
| PL/IG | .PL1G |
| PL/P | .PLP |
| PMA | .PMA |
| COBOL | .COBOL |
| Fortran | .FTN |
| Fortran 77 | .F77 |
| Pascal | .PASCAL |
| Basic | .BASIC |
| Design | .DES |
| CPL | .CPL |
| LISP | .LISP |
| EMACS Extension File | .EMACS |

The formats of each of these file types are described in a series of appendices.
These are very important as particular character sequences are generated for a given file type.

# 8 REFORM

REFORM is a REpresentation FORMatter for use on files containing STROMA constructs embedded within Design Blocks.

## 8.1 Function

REFORM is intended to speed STROMA design entry and verification through three functions:

1. Elementary syntax checking of STROMA constructs.

2. Reformatting the STROMA design for increased readability through uniform indentation conventions.

3. Simple consistency checking over the design, flagging the missing design of invoked units and the inclusion of uninvoked design units.

Use of REFORM allows the programmer to enter STROMA design quickly, without regard to format, and yet still have readable designs whose physical formats reflect their logical structures. STROMA designs which have already been indented by REFORM are passed through this formatter without change, allowing easy editing of existing designs.

## 8.2 User Interface

REFORM is invoked by:

```
REFORM input [output] [control arguments]
```

Where 'input' is the treename of the source file to be reformated and 'output' is the treename of the result file, if omitted the source file will be replaced. If errors are detected, the input file will not be modified and the output will be left in a temporary file whose name will be given to the user in an error message.

The control arguments may be chosen from the following in any order:

| | |
|---|---|
| -NO_UPPER_CASE, -NUC | Inhibits the conversion of keywords to upper case. |
| -UPPER_CASE_LABELS, -UCL | Causes labels to be converted to upper case. (Default - labels output in the form read.) |

| | |
|---|---|
| -NO_QUERY, -NQ | Suppresses verification request if 'output' already exists or is omitted. (Default - verification will be requested.) Ignored if no treenames have been specified, the user needs help. |

The filename selected via the input treename must conform to the standard naming convention adopted by the S.P.S. package. This limits REFORM to only processing files whose language format is indicated using a file suffix.
The recognised suffixes are:

| contents | suffix |
|---|---|
| PL/I | .PL1 |
| PL/IG | .PL1G |
| PL/P | .PLP |
| PMA | .PMA |
| COBOL | .COBOL |
| Fortran | .FTN |
| Fortran 77 | .F77 |
| Pascal | .PASCAL |
| Basic | .BASIC |
| Design | .DES |
| CPL | .CPL |
| LISP | .LISP |
| EMACS Extension File | .EMACS |

In addition to the above, REFORM accepts a null suffix as indicating a Design file.

If an output file is specified it must conform to the standard and be of the same language type as the input file.

The formats of each of these file types are described in a series of appendices. These are very important as particular character sequences are recognised/generated for a given file type, and particular character sequences are discarded. If the described formats are not used it is possible that a file may not be formatted.

## 8.3 Processing

### 8.3.1 Source format

The design blocks present in the input file must adhere to the layout requirement of the language contained in the file, together with the further restrictions imposed by S.P.S. (described in the appendices).

Abbreviations have been defined for the STROMA keywords as follows:

| keyword | abbreviation |
|---|---|
| REPEAT | REP |
| END-REPEAT | ER |
| BREAK | BRK |
| SELECT | SEL |
| END-SELECT | ES |
| MONITOR | MON |
| END-MONITOR | EM |
| ADMIT | ADM |
| ELSE | OTHERWISE |

These are recognised on input as being equivalent to the corresponding keyword and converted to the corresponding keyword.

The actual design expressed in STROMA has only two simple, restrictions other than syntactic restrictions, applied to it.

1. All STROMA keywords, except for WHILE and UNTIL, must be the first word on a line in order for REFORM to recognise them. The keywords FOR, WHILE and UNTIL are considered to be extensions of REPEAT.

2. A STROMA construct must not span more than one design block.

Consequently the STROMA contained in a design block can be free format subject to the two restrictions mentioned above.

### 8.3.2 Output format

REFORM performs simple indentation together with some text manipulation on the STROMA source. The indentation performed is fixed but the text manipulation is selected via the command line keywords.

The indentation rules applied are:

1. Begin construct and end construct keywords are aligned, in addition the keywords identifying subordinate WHEN and ELSE clauses of the SELECT construct are aligned with the SELECT keyword.

2. Text within a construct is indented one level (three spaces).

3. Blank lines are maintained, none are generated.

4. Lines commencing with a fullstop ('.') are left unchanged as these may represent Runoff commands embedded in the Design.

The text manipulation performed is the forcing of recognised keywords or labels to uppercase, if required, and the detection of '_' as the first character on a line. The single underscore character is translated into seven underscores to cause the line to be indented further. This can be used as a means of indicating text that is

subordinate to, or a continuation of, the preceding line.

### 8.3.3 Checking

#### 8.3.3.1 Syntax Checking

Syntax checking is performed on each design block in isolation with END-DESIGN as the closing keyword of the grammar. The syntax rules applied are those described informally in the section on STROMA. If a syntax error is detected in a design block then the rest of that design block is not parsed, it is just copied to the output file. REFORM restarts parsing design on the next design block encountered.

Syntax errors are reported with the following error message format:

> Error at line 'line no.' in design 'clause' starting on line 'line no.' contains an unexpected 'keyword' at line 'line no.'.

This indicates both the current construct and the illegal keyword detected within it.

#### 8.3.3.2 Consistency Checking

REFORM performs simple consistancy checking over the design it processes. The checking is only performed on syntactically correct design.

The objective of the consistancy checking is to notify the user of REFORM when no design exists within the file for an invoked unit and/or when a design exists for an invokable unit but it is not referenced within the rest of the design. This results in two benefits: one, complete designs can be detected; and two, attention can be drawn to invoked units whose design is external to the design being processed.

Therefore a complete and error free design when processed by REFORM will result in REFORM only notifying the user of external units referenced within the design. This facility can be used to indicate PRIMOS routines referenced in a design. Within the design they can be included as invoked units but of course no corresponding design will be present. When the design is processed by REFORM the routines present will be flagged as invoked routines with no design.

The consistancy checking is performed by REFORM building two internal tables, one of invoked routine names and the other of named design units. After formatting the design contained in a file, provided no errors were detected, these two tables are compared and any discrepancies are reported. The text following the keyword DO upto a ':' or '(' or '[' character is used as the name of an invoked routine, and the label preceding an invoked design unit as the name of a design unit.

# 9 RESTATE

RESTATE is a REpreSenTATion convErter.

## 9.1 Function

RESTATE converts a file containing Title Lines, Description Blocks and Design Blocks into a comment form suitable for the intended implementation language.

This relieves the user of the tedious task of converting the contents of a design file into a form compatible with the commenting requirements of the implementation language.

As a by-product a file containing comments in any of the recognised languages may be converted to a file of another type. (Note that RESTATE does not change any code statements and will therefore not convert a source program from one language to another.)

## 9.2 User Interface

RESTATE is invoked by:

```
RESTATE input [output] [control arguments]
```

Where 'input' is the treename of the source file to be input for comment conversion, and 'output' is the treename of the file to be produced. If 'output' is omitted and no control argument is supplied to specify the output file type, then the input file will be replaced. If errors are detected, the input file will not be modified and the output will be left in a temporary file whose name will be given to the user in an error message.
If no errors are detected and the input file is not being overwritten then the input file will be deleted.

The program will only perform its conversion between files whose names include recognised suffixes.

The control arguments may be chosen from the following in any order:

| | |
|---|---|
| -NO_QUERY, -NQ | Suppresses verification request if 'output' already exists or is omitted. Also suppresses verification request on possible deletion of input file. (Default - verification will be requested.) |
| -xxx | This parameter may be one of the strings: 'PL1' 'PLP' 'PL1G' 'FTN' 'F77' 'PMA' 'COBOL' 'PASCAL' 'DES' 'BASIC' 'CPL' 'LISP' 'EMACS' and indicates that the file to be |

produced should have a similar name to that of the input, but with the last suffix replaced by 'xxx'.
If this control argument is specified then the output file should not be specified.

## 9.3 Processing

### 9.3.1 Input File Processing

This tool will only process a file with an acceptable suffix.

The following table links the format of the contents of a file to its corresponding suffix:

| contents | suffix |
|---|---|
| PL/I | .PL1 |
| PL/IG | .PL1G |
| PL/P | .PLP |
| PMA | .PMA |
| COBOL | .COBOL |
| Fortran | .FTN |
| Fortran 77 | .F77 |
| Pascal | .PASCAL |
| Basic | .BASIC |
| Design | .DES |
| CPL | .CPL |
| LISP | .LISP |
| EMACS Extension File | .EMACS |

The formats of each of these file types are described in a series of appendices.
These are very important as particular character sequences are recognised for a given file type, and particular character sequences are discarded. If the described formats are not used it is possible that an incorrect conversion will be performed.

### 9.3.2 Output File Processing

This tool will only process a file with an acceptable suffix.

The following table links the format of the contents of a file to its corresponding suffix:

| contents | suffix |
|---|---|
| PL/I | .PL1 |
| PL/IG | .PL1G |
| PL/P | .PLP |
| PMA | .PMA |

| | |
|---|---|
| COBOL | .COBOL |
| Fortran | .FTN |
| Fortran 77 | .F77 |
| Pascal | .PASCAL |
| Basic | .BASIC |
| Design | .DES |
| CPL | .CPL |
| LISP | .LISP |
| EMACS Extension File | .EMACS |

The formats of each of these file types are described in a series of appendices.
These are very important as particular character sequences are generated for a given file type.

When the suffix of the input file is not 'DES' the comment structure of the input file is maintained in the output file.

When the suffix of the input file is 'DES' then the following comment structure is created, as appropriate to the commenting conventions required for the file format:

Title Line - a single line comment
Description Block - a block comment
Design Block - a block comment
Start Code Line - a single line comment
End Code Line - a single line comment

# 10 INFORM

INFORM is an INstruction FORMatter for use with PL/P programs.

## 10.1 Function

INFORM is intended to speed PL/P program entry and development through two functions:

1. Pre-compilation syntax checking for matching parentheses, ends, quotes, if-then-else constructs, and comment delimiters.

2. Reformatting the program text for increased readability through uniform spacing and indentation conventions.

Use of INFORM allows the programmer to enter PL/P programs quickly, without regard to format, and yet still have readable programs whose physical formats reflect their logical structures. Programs which have already been indented by INFORM are passed through the formatter without change, allowing easy editing of existing programs.

## 10.2 User Interface

INFORM is invoked by:

INFORM input [output] [control arguments]

Where 'input' is the treename of the source file to be indented and 'output' is the treename of the result file, if omitted the source file will be replaced. If errors are detected, the input file will not be modified and the output will be left in a temporary file whose name will be given to the user in an error message.

The control arguments may be chosen from the following in any order:

| | |
|---|---|
| -LMARGIN xx, -LM xx | Sets the indentation for the outermost level of nesting to be 'xx' spaces (default 8, range: 1 to right margin minus 39). |
| -RMARGIN xx, -RM xx | Sets the column number beyond which non-comment, non-string text will not be placed to be 'xx' (default 98, the size of the PL/P listing without '-offset', range: left margin + 39 to 256). |
| -COMMENT_COL xx, -CC xx | Sets the column to which remark comments will be indented to be 'xx' (default 50, range: left margin + 14 to right margin - 25). |
| -INDENT xx, -IND xx | Sets the number of spaces to indent for |

| | |
|---|---|
| | each logical level of nesting to be 'xx' (default 3). |
| -NO_FILL, -NF | Causes format to maintain the line structure of the input, except extra lines will be added as necessary for line breaking. (default 'fill', line breaks within input statements will be ignored, except blank lines are maintained). |
| -NO_QUERY, -NQ | Suppresses verification request if 'output' already exists or is omitted (default verification will be requested). |

## 10.3 Processing

### 10.3.1 Limitations

INFORM cannot process a statement longer than 8191 characters or 1000 lexical items, excluding labels and preceding header comments (this limitation does not apply to declaration statements). All the header comments preceding a statement may not contain more than 8191 characters. No line of text, after indentation, may be longer than 256 characters.

### 10.3.2 Character-level Processing

INFORM deletes unnecessary spaces within lines and ensures that '&', '=', '/', ' ', '*', '<', '>', '^=', '^>', '^<', ' ', '**', '<=', '>=', and '->' are both preceded and followed by a blank, that ',' and ')' are followed by a blank when that makes sense (e.g., '))', not ') )'), that '(' and '^' are preceded by a blank when that makes sense, and that '+' and '-' are always preceded by a blank and are followed by a blank when they are not unary. Needless to say, this processing does NOT occur within comments and character string constants.

### 10.3.3 Comment Processing

INFORM recognizes two kinds of comments: remark comments, which are those preceded on the input line by one or more non-blank characters, and header comments, which are those on a line by themselves.

Remark comments are aligned to the column specified by the '-cc' command line parameter, or 3 columns to the right of the last text on the line, whichever is larger. Text following the remark is placed on a new line.

Header comments are left-justified and a blank line is inserted before and after, if one is not already present.

INFORM manipulates only the spaces preceding the comment text, all other internal spaces are preserved. For both header and remark comments spaces following the start comment symbol '/*' are compressed to one space.
Thus:

```
/*       A line of text.
```

is transformed into

```
/* A line of text.
```

Spaces following the text of a comment and the close comment symbol '*/' are unaltered, except that if no spaces are present one is inserted. A close comment symbol on a line alone is aligned with its corresponding start symbol.

For comment continuation lines INFORM attempts to perform simple indentation. Continuation line of remark comments are left justified. Thus:

```
............/* The first line of remark
the second line and now
the last line */
```

becomes

```
............/* The first line of a remark
               the second line and now
               the last line */
```

A similar approach is taken to header comments, and in the simple case left justified text is produced. However as some spaces at the start of lines may have been introduced by the comment's author an attempt is made to preserve them. The approach taken is that as three spaces are necessary to give left justified text, INFORM ensures that at least three are present. Three or less spaces on a line are forced to three spaces, more than three spaces are left unaltered.

The following is an example of INFORMs handling of header comments:

```
                /* A header comment with
continuation lines, preceded by a varying
 number of spaces. (1 space becomes 3)
  A continuation line, (2 spaces become 3)
   another line (3 spaces remain as 3)
    penultimate line (4 spaces remain as 4)
      last line (6 spaces remain as 6) */
```

becomes

```
/* A header comment with
   continuation lines, preceded by a varying
```

```
number of spaces. (1 space becomes 3)
A continuation line, (2 spaces become 3)
another line (3 spaces remain as 3)
 penultimate line (4 spaces remain as 4)
   last line (6 spaces remain as 6) */
```

### 10.3.4 Label Processing

Each label (with its associated remark comment, if any) is placed on a separate line. The label is left-justified, regardless of current indentation level, for ease of location when scanning the text.

### 10.3.5 Statement Processing

Each statement begins a new line in the indented text, with the starting column determined as described below by its relationship with 'do', 'proc', 'begin', 'select', 'end', and 'if-then-else' statements. If the statement after indentation is larger than the right margin will allow on a single line, INFORM attempts to break the line at a delimiter or, failing that, just before the overflowing string, identifier, or number. If a single string or identifier is too large to fit between the indented margin and the right margin, the right margin is ignored (as it is for comments). The rest of the text will continue on the next line indented an additional 3 increments past the current logical nesting level.

Text contained in 'proc', 'begin', 'do', and 'select' groups (along with the 'proc', but not including the other 3 statements) is indented one increment past the surrounding text; 'end' statements are aligned with the rest of the group they close and cause the following statement to be 'outdented' one increment.

The 'then' clause of an 'if' statement is placed on a separate line and aligned with the 'if' (a 'do', 'begin', or 'select' group in a 'then' clause is indent one increment past the 'then'); similarly, if present, the 'else' is aligned to the level of the 'if' (a 'do', 'begin', or 'select' group in an 'else' clause is indented one increment past the 'else' level).

The 'when' and 'otherwise' statements of a 'select' statement are aligned with their controlling 'select'. The contained statement is indented one increment on a seperate line.

### 10.3.6 Declaration and '%statement' Processing

The first line is left-justified and subsequent lines, unless resulting from a broken line, are indented 4 spaces if 'dcl' was used, 9 for '%replace' constructs, and 8 if 'declare' was the keyword, so that the identifiers will line up. No special processing occurs within parentheses in a declaration statement. The text is passed directly through to the output, subject to line breaking and the appropriate

indentation. At parenthesis level 0, however, two functions occur:

1. Each comma results in a new line, enforcing the 'one identifier per line' constraint of structured coding (which allows one to scan for the declaration of an identifier more easily).

2. Structures are indented one increment for each level greater than 1.

## 11 Warnier Diagrams

One technique that it is felt can be used more and more in the future is that known as the Warnier or the Warnier/Orr System.

This technique is particularly useful as it can be used to express both the data and the logic flow within a system, without significantly constraining a design.

Unfortunately where these are used the information is rarely maintained and is frequently discarded. In order to obtain the best possible return from the use of this technique a machine interface must be constructed.

Further information on this technique can be obtained from the book:
Structured Systems Development
available in the R&D (UK) Library,
or from John Howell

### 11.1 REWARD

A tool is required to facilitate the inputing of Warnier diagram information, layout the information and output sections of the design.

This tool has not been implemented, nor has its required functionality been determined.

## 12 SDL

SDL (Software Design Language) is a tool to aid in designing and documenting a program or system of programs.

This is described in PE-T-462.

This package does not appear to encourage/allow the design and the code of a program to exist in the same file.
It also requires a number of control commands to be included within the design.

The software is available on the Bedford machine as X.SDL.
Instructions for using it are available in PE-T-462.

# 13 DENOTE

## 13.1 Function

DENOTE - the DEsign NOTEbook builder - is a utility that has been designed to facilitate the production of a piece of PRIME Internal project documentation, hereafter referred to as the Design Notebook.

The purpose of DENOTE is to extract Description Blocks and/or Design Blocks from files in a suitable format, and to produce an amalgam of these blocks, together with Runoff control commands for subsequent input to Runoff.

The existence of this tool makes it very simple to produce an up-to-date document at any stage of a project's life cycle from which an assessment can be made of the state of the project.

It should be noted that it is essential that some ground rules be established from day 1 as regards to the manner in which Description Blocks and Design Blocks are to be created, since this will affect the subsequent appearance of the Design Notebook. Any need to perform substantial editing of these blocks will detract from the ease with which the Design Notebook can be produced. It is also important that the contents of these blocks be considered at a project level.

## 13.2 User Interface

DENOTE is invoked by:

```
DENOTE input output [control arguments]
```

Where 'input' is the treename of the source file from which information is normally (see -LIST option) extracted, and 'output' is the treename of the file to be produced. If either of these names are omitted the program will request that these names be provided before it will continue.

'Input' may in fact be a wildcarded name, thus causing the program to perform its extraction from a number of files. The program will only perform information extraction from files whose names include a recognised suffix.

The control arguments may be chosen from the following in any order:

| | |
|---|---|
| -LIST, -LI | If this is specified, 'input' is constrained to be a simple (not wildcarded) treename. 'Input' must then contain a list of treenames (optionally wildcarded) from which information is to be extracted. This is a specific use of a Project |

Catalogue, described earlier.
This option facilitates the production of:
1. a partial Design Notebook
2. an ordered Design Notebook
3. a Design Notebook when the filenames to be specified cannot easily be expressed by a single wildcarded name

-DESCRIPTION, -DSC

If this is specified, only Description Blocks will be extracted from the inputs.
If neither -DESCRIPTION nor -DESIGN are supplied as control arguments then both Description Blocks and Design Blocks will be extracted.

-DESIGN, -DGN

If this is specified, only Design Blocks will be extracted from the inputs.
If neither -DESCRIPTION nor -DESIGN are supplied as control arguments then both Description Blocks and Design Blocks will be extracted.

-ADJUST, -ADJ

If this is specified 'output' will be a Runoff compatible file produced in 'adjust' mode.
If this is not specified (default) 'output' will be a Runoff compatible file produced in 'no fill' mode.

-NO_QUERY, -NQ

If this is specified and 'output' already exists then the program will overwrite the file without requesting permission to do so.

-WIDTH x, -W x

If this is specified then a line width of 'x' is created in the output file, otherwise a line width of 85 is generated.
'x' is required to be greater than 14 and to not exceed 170.

-BLANK x, -BL x

If this is specified then 'x' should be a charcter that will be created and used as the Runoff 'required' blank character when in 'ADJUST' mode.
If this is not specified the character '&' is used.

-INFORM_SPLIT, -IS

If this is specified then DENOTE will report any input lines that it has to split when not in 'ADJUST' mode.

| | |
|---|---|
| -NO_MESSAGE, -NM | If this is specified warning messages are output to a temporary file instead of to the screen. |

## 13.3 Processing

### 13.3.1 Project Catalogue Processing

DENOTE recognises the following types of entries in a Project Catalogue:

1. Runoff entry.
   This has the format:
   RUNOFF <Runoff command>
   For further details see later in this document.

2. File entry.
   This has the format:
   treename [control arguments]
   The control arguments recognised by DENOTE are:

| | |
|---|---|
| -NO_BOOK | This indicates that there is to be no entry in the Design Notebook for the indicated file |

   Any further control arguments are assumed to relate to project utilities and are ignored by DENOTE.

### 13.3.2 Input File Processing

This tool will only extract information from a file with an acceptable suffix. Since the extraction method is linked to this suffix it is important that the routine format selected corresponds to the file suffix.

The following table links the format of the contents of a file to its corresponding suffix:

| contents | suffix |
|---|---|
| PL/I | .PL1 |
| PL/IG | .PL1G |
| PL/P | .PLP |
| PMA | .PMA |
| COBOL | .COBOL |
| Fortran | .FTN |
| Fortran 77 | .F77 |
| Pascal | .PASCAL |
| Basic | .BASIC |
| Design | .DES |
| CPL | .CPL |
| LISP | .LISP |
| EMACS Extension File | .EMACS |

The formats of each of these file types are described in a series of appendices.
These are very important as particular character sequences are recognised for a given file type, and particular character sequences are discarded. If the described formats are not used it is possible that information could be discarded by DENOTE.

At the level of an individual file the only processing performed is to cause the outputting of its treename in a hyphenated box within the main part of the document and the generation of a first-level title in the Table of Contents.

### 13.3.3 Title Processing

The normal format of a 'TITLE' line is:

TITLE: <name>

as defined in the section on Routine Format.

The output processing performed specific to a 'TITLE' line is to cause the outputting of the name in an asterisked box (on a new page if appropriate) within the main part of the document and the generation of a second-level title in the Table of Contents.

If a Description Block or a Design Block is being processed when the 'TITLE' line is met then that processing is terminated.

Since it is accepted that a user may require more than 2 levels of titleing in a Table of Contents some additional forms of the 'TITLE' line are accepted, as follows:

TITLE-D: <name>
Causes the title-level to be increased by 1 before outputting the title to the Table of Contents.
If the simple 'TITLE' line is used after this command then that title is output at the level which is then current.

TITLE-U:
Causes the title-level to be decremented by 1. The title-level is never decremented past 2.

TITLE-U: <name>
Causes the title-level to be decremented by 1 before outputting the title to the Table of Contents.

### 13.3.4 Description Processing

The normal format of a Description Block is:

```
START-DESCRIPTION:  [<name>]
     block of text
END-DESCRIPTION
```

as defined in the section on Routine Format.

If the start of a Description Block is located and it is to be extracted then commands are generated to ensure that at least 10 lines will fit on the current page.
The generation of the heading:

DESCRIPTION: [<name>]

is then caused (note that in this case 'name' is optional). The lines of the block are then passed across to the output file.

When DENOTE is not in 'ADJUST' mode the only processing performed on these lines relates to their length. If the length of a line exceeds the line width then DENOTE attempts to split the contents of the line in a reasonable manner (without this, Runoff would arbitrarily split the line). Any line splitting is reported if this is requested.

When DENOTE is in 'ADJUST' mode then it additionally analyses the contents of each line to determine whether any suplementary Runoff commands should be included to generate a layout consistent with that input. At a simple level this consists of generating commands to control indentation and spacing.

At a further level is the need for the inclusion of mandatory spaces when a tabular layout is required and the need to cause line breaking to be performed without including blank lines. Each of these tasks, require some indication in the text of the user's requirement. The character ascii-200 has been chosen for each of these requirements and must be included in the Description Block by the user to obtain the desired effect.
This character was chosen because it is not printed by the spooler. It can be obtained on many terminals by the character combination control and '@'.
The presence of ascii-200 prefixing a group of spaces causes the 'required' space character to be generated for each of these spaces, thus causing Runoff to force the correct number of spaces.
The presence of ascii-200 at the end of a line causes a line break command to be generated.

### 13.3.5 Design Processing

The normal format of a Design block is:

```
START-DESIGN:  [<name>]
     block of text
END-DESIGN
```

As defined in the section on Routine Format.

If the start of a Design Block is located and it is to be extracted then commands are generated to ensure that at least 10 lines will fit on the current page.

The generation of the heading:

DESIGN: [<name>]

is then caused (note that in this case 'name' is optional).

The lines of the block are then passed across to the output file.

When DENOTE is not in 'ADJUST' mode the only procesing perfomed on these lines relates to their length. If the length of a line exceeds the line width then DENOTE attempts to split the contents of the line in a reasonable manner (without this, Runoff would arbitrarily split the line). Any line splitting is reported if this is requested.

When DENOTE is in 'ADJUST' mode each line is output in such a way as to appear on a new line with appropriate indentation. There is no need for the inclusion of special characters.

### 13.3.6 Runoff Command Embedding by the User

The user of DENOTE may wish to cause additional Runoff commands to be fed through to the output file for some purpose.
This can be done in two ways.

Runoff commands may be embedded in the Description Blocks and Design Blocks. In this case the '.' prefixing the command must be the first character on a line, after any comment symbol required by the file format.

Runoff commands may also be embedded in the input file when the 'LIST' option is used. In this case an entry in the file should have the format:

RUNOFF command

If either of these methods is used to include additional formatting commands, care should be taken not to disturb the formatting performed by DENOTE.
DENOTE does in fact detect and interpret a number of Runoff commands, that directly impact on its own formatting, as follows:

| | |
|---|---|
| .WIDTH x,.W x | Causes DENOTE to reset the line width created in the output file. 'x' is required to be greater than twice the margin size, and to not exceed 170 |
| .SMARGIN x,.SM x | Causes DENOTE to reset the side margin size in the output file 'x' is required to be such that twice its |

| | |
|---|---|
| | value does not exceed the width currently in force |
| .BLANK x, .BL x | Causes DENOTE to change the value of the 'required' blank character to 'x' |

All other Runoff commands are simply passed through to the output file,

Note that although DENOTE interprets these commands and passes them through to the output file, these are not changed for the Table of Contents.
The line width, side margin size and 'required' space character used in the Table of Contents are those in force when the program starts execution.

## 13.4 Runoff Considerations

DENOTE requires to issue a number of general Runoff commands to ensure that the Design Notebook is output in the intended manner.

Most of these general commands are located near the beginning of the output file:

- decimalisation setting
- footer initialisation
- page width setting
- table of contents initialisation
- '.nfill' - if DENOTE is not in ADJUST mode
- 'required' blank initialisation

and near the end of the output file:

- '.fill' if DENOTE is not in ADJUST mode
- '.adj' if DENOTE is not in ADJUST mode
- table of contents closure
- table of contents insertion into output file

If the output from DENOTE is to be amalgamated with another document some additional commands may require to be inserted.

It is also important that lines of description or design do not commence with a full-stop unless they are intended to be Runoff commands.

## 14 Design Notebook

This document should be produced in every project and this should be planned from day 1 of a project's life.

This document originates within R&D, is developed in parallel with the project, and on completion is 'shipped' with the product to the same audience as its source.

This document must contain:

1. Any global information belonging to a project such as:
   high level design information
   data structure descriptions
   naming conventions used
   non-standard design technique documentation

2. Information on every module:
   description
   design

The purpose of this document is to maintain in one place the detailed design of a product and all information gathered during the development.
This can be of significant interest to all team members during the development of a product, and subsequently to any people assigned to maintain it. (The information may also assist field analysts if circulated that far.)

The production of this document at the end of a project is a prohibitive and unsatisfying task. This is both in terms of the sheer size of the task, and the effect on project members of having to produce this in retrospect.

For this reason, among others, it is important that the need for this document be accepted from day 1 of a project. Most of the information that belongs in the Design Notebook should exist within the various modules from their inception, within their Description Blocks and Design Blocks.

By adopting the recommmended routine format from day 1 of a project's lifecycle, this document can easily be produced automatically by using DENOTE.

This document should be produced at regular intervals by the project leader to facilitate an up-to-date assessment of a projects status, as input to Walkthrough sessions, and for major reviews.

## 15 Other Areas for Consideration

After the initial polling of the members of R&D (UK) a list was produced itemising those areas that appeared to require attention. This list appears below:

1. Definition of Design Techniques required.
   A number of books have been ordered for the R&D (UK) Library.

2. A checklist required for project control.

3. The definition of a recommended pseudo language.
   The definition of STROMA has been produced.
   No correspondance between STROMA constructs and programming language constructs has been suggested.

4. Standards for program layout required at file level and program level.
   Recomendations required on program commenting and indentation.
   A Routine Format has been suggested, and a PL/P program formatter has been produced.

5. Definition required of the documents required at all stages of a project's lifecycle, with respect to title and contents.

6. Guidelines requested for the use of commonly used programming languages, in terms of both style and efficiency.

7. Naming conventions requested for files, routines, etc.

8. Definition and design of tools requested to:
   a) Format PL/P programs - INFORM
   b) Program design aid
   c) Program commenting aid
   d) Project documentation aid - DENOTE
   e) Source File System

9. Creation and maintenance of libraries for utilities, source routines and declarations.

10. Organisation of on-line development by means of ufd structuring and the use of the source file system

It has not been possible to attempt to do something about all of these items.
Decisions remain to be made as to which of the above should be followed up, and what resources can be made available for this.

## Appendix A - Routine Format for Design files

The following indicates how the Routine Format described earlier should appear in a design file.
Since this file format has no language related formatting constraints, no additional characters are included.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
```

May be preceeded by any number of spaces.

The format is:

```
TITLE: the identity of the routine

START-DESCRIPTION: [ <name> ]
This is a block of narrative describing the function of the routine.
Any spaces preceeding the lines in this block are significant.
END-DESCRIPTION

START-DESIGN: [ <name> ]
This is a block of design information.
Any spaces preceeding the lines in this block are significant.
END-DESIGN
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

## Appendix B - Routine Format for PL/I files

The following indicates how the Routine Format described earlier should appear in a PL/I, PL/P or PL/IG file.
The information held in this file must be compatible with the commenting requirements of the PL/I languages.
It must also be compatible with the constraints imposed on the format of block comments by the comment handling of INFORM (and this has influenced the design of the other SPS tools).

The SPS package processes comment blocks that commence with a comment start symbol in column 1. (INFORM will in fact recognise and process comments that appear anywhere in a line.) If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.
In the case of any lines that continue such a comment, upto three spaces at the beginning of the line are considered to be included for cosmetic reasons. These are removed on input and forced on output.

Note that the Description Block and Design Block below are block comments.
An '&' is used below to represent any space characters generated by the SPS package.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the '/*&' or '&&&' begining its line by any number of spaces.

The format is:

```
/*& TITLE: the identity of the routine */

/*& START-DESCRIPTION: [ <name> ]
&&& This is a block of narrative describing the function
&&& of the routine.
&&& This format will be generated by any of the tools in the
&&& SPS suite.
&&& END-DESCRIPTION */

/*& START-DESIGN: [ <name> ]
&&& This is a block of design information.
&&& This format will be generated by any of the tools in the
&&& SPS suite.
&&& END-DESIGN */
```

```
/*& START-CODE: */
The program
/*& END-CODE */
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

## Appendix C - Routine Format for Fortran files

The following indicates how the Routine Format described earlier should appear in a Fortran file.
The information held in this file must be compatible with the commenting requirements of the Fortran language.

If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.

Note that the Description Block and Design Block below are line comments.
An '&' is used below to represent any space characters generated by the SPS package.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the 'C&' beginning its line by any number of spaces.

The format is:

```
C& TITLE: the identity of the routine

C& START-DESCRIPTION: [ <name> ]
C& This is a block of narrative describing the function
C& of the routine.
C& This format will be generated by any of the tools in the
C& SPS suite.
C& END-DESCRIPTION

C& START-DESIGN: [ <name> ]
C& This is a block of design information.
C& This format will be generated by any of the tools in the
C& SPS suite.
C& END-DESIGN

C& START-CODE:
The program
C& END-CODE
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

## Appendix D - Routine Format for PMA files

The following indicates how the Routine Format described earlier should appear in a PMA file.
The information held in this file must be compatible with the commenting requirements of the PMA language.

If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.

Note that the Description Block and Design Block below are line comments.
An '&' is used below to represent any space characters generated by the SPS package.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the '*&' beginning its line by any number of spaces.

The format is:

```
*& TITLE: the identity of the routine

*& START-DESCRIPTION: [ <name> ]
*& This is a block of narrative describing the function
*& of the routine.
*& This format will be generated by any of the tools in the
*& SPS suite.
*& END-DESCRIPTION

*& START-DESIGN: [ <name> ]
*& This is a block of design information.
*& This format will be generated by any of the tools in the
*& SPS suite.
*& END-DESIGN

*& START-CODE:
The program
*& END-CODE
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

## Appendix E - Routine Format for COBOL files

The following indicates how the Routine Format described earlier should appear in a COBOL file.
The information held in this file must be compatible with the commenting requirements of the COBOL language.

If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.

Note that the Description Block and Design Block below are line comments.
An '&' is used below to represent any space characters generated by the SPS package.
Note that '$' is used below to represent a mandatory character, thus the '*' will appear in position 7 of a line.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the '$$$$$$*&' beginning its line by any number of spaces.

The format is:

```
$$$$$$*& TITLE: the identity of the routine

$$$$$$*& START-DESCRIPTION: [ <name> ]
$$$$$$*& This is a block of narrative describing the
$$$$$$*& function of the routine.
$$$$$$*& This format will be generated by any of the tools in the
$$$$$$*& SPS suite.
$$$$$$*& END-DESCRIPTION

$$$$$$*& START-DESIGN: [ <name> ]
$$$$$$*& This is a block of design information.
$$$$$$*& This format will be generated by any of the tools in the
$$$$$$*& SPS suite.
$$$$$$*& END-DESIGN

$$$$$$*& START-CODE:
The program
$$$$$$*& END-CODE
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that

of the Description Block and the Design Block.

## Appendix F - Routine Format for Pascal files

The following indicates how the Routine Format described earlier should appear in a Pascal file.
The information held in this file must be compatible with the commenting requirements of the Pascal language.

The SPS package processes comment blocks that commence with a comment start symbol in column 1.
If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.
In the case of any lines that continue such a comment, upto two spaces at the beginning of the line are considered to be included for cosmetic reasons. These are removed on input and forced on output.

Note that the Description Block and Design Block below are block comments.
An '&' is used below to represent any space characters generated by the SPS package.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the '{&' or '&&' beginning its line by any number of spaces.

The format is:

```
{& TITLE: the identity of the routine }

{& START-DESCRIPTION: [ <name> ]
&& This is a block of narrative describing the function
&& of the routine.
&& This format will be generated by any of the tools in the
&& SPS suite.
&& END-DESCRIPTION }

{& START-DESIGN: [ <name> ]
&& This is a block of design information.
&& This format will be generated by any of the tools in the
&& SPS suite
&& END-DESIGN }

{& START-CODE: }
The program
{& END-CODE }
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

## Appendix G - Routine Format for Basic files

The following indicates how the Routine Format described earlier should appear in a Basic file.
The information held in this file must be compatible with the commenting requirements of the Basic language.

If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.

Note that the Description Block and Design Block below are line comments.
An '&' is used below to represent any space characters generated by the SPS package.
Note that '$' is used below to represent a mandatory character, thus the 'REM' will appear in position 6 of a line. (This allows the inclusion of a line number of upto five digits.)

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the '$$$$$REM&' beginning its line by any number of spaces.

The format is:

```
$$$$$REM& TITLE: the identity of the routine

$$$$$REM& START-DESCRIPTION: [ <name> ]
$$$$$REM& This is a block of narrative describing the
$$$$$REM& function of the routine.
$$$$$REM& This format will be generated by any of the tools in the
$$$$$REM& SPS suite.
$$$$$REM& END-DESCRIPTION

$$$$$REM& START-DESIGN: [ <name> ]
$$$$$REM& This is a block of design information.
$$$$$REM& This format will be generated by any of the tools in the
$$$$$REM& SPS suite.
$$$$$REM& END-DESIGN

$$$$$REM& START-CODE:
The program
$$$$$REM& END-CODE
```

If a History Block is created manually (or maintained by the Source

File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

## Appendix H - Routine Format for CPL files

The following indicates how the Routine Format described earlier should appear in a CPL file.
The information held in this file must be compatible with the commenting requirements of the CPL language.

If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.

Note that the Description Block and Design Block below are line comments.
An '&' is used below to represent any space characters generated by the SPS package.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the '/*&' beginning its line by any number of spaces.

The format is:

```
/*& TITLE: the identity of the routine

/*& START-DESCRIPTION: [ <name> ]
/*& This is a block of narrative describing the function
/*& of the routine.
/*& This format will be generated by any of the tools in the
/*& SPS suite.
/*& END-DESCRIPTION

/*& START-DESIGN: [ <name> ]
/*& This is a block of design information.
/*& This format will be generated by any of the tools in the
/*& SPS suite.
/*& END-DESIGN

/*& START-CODE:
The program
/*& END-CODE
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

## Appendix I - Routine Format for LISP files

The following indicates how the Routine Format described earlier should appear in a LISP file.
The information held in this file must be compatible with the commenting requirements of the LISP language.

If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.

Note that the Description Block and Design Block below are line comments.
An '&' is used below to represent any space characters generated by the SPS package.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the ';&' beginning its line by any number of spaces.

The format is:

```
;& TITLE: the identity of the routine

;& START-DESCRIPTION: [ <name> ]
;& This is a block of narrative describing the function
;& of the routine.
;& This format will be generated by any of the tools in the
;& SPS suite.
;& END-DESCRIPTION

;& START-DESIGN: [ <name> ]
;& This is a block of design information.
;& This format will be generated by any of the tools in the
;& SPS suite.
;& END-DESIGN

;& START-CODE:
The program
;& END-CODE
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

## Appendix J - Routine Format for EMACS files

The following indicates how the Routine Format described earlier should appear in a EMACS file.
The information held in this file must be compatible with the commenting requirements of the EMACS language.

If the comment start symbol is followed by a space then this is considered to be an extension of the comment symbol (to aid legibility). This space is removed on input and forced on output.

Note that the Description Block and Design Block below are line comments.
An '&' is used below to represent any space characters generated by the SPS package.

Each of the words:

```
TITLE
START-DESCRIPTION
END-DESCRIPTION
START-DESIGN
END-DESIGN
START-CODE
END-CODE
```

may be separated from the ';&' beginning its line by any number of spaces.

The format is:

```
;& TITLE: the identity of the routine

;& START-DESCRIPTION: [ <name> ]
;& This is a block of narrative describing the function
;& of the routine.
;& This format will be generated by any of the tools in the
;& SPS suite.
;& END-DESCRIPTION

;& START-DESIGN: [ <name> ]
;& This is a block of design information.
;& This format will be generated by any of the tools in the
;& SPS suite.
;& END-DESIGN

;& START-CODE:
The program
;& END-CODE
```

If a History Block is created manually (or maintained by the Source File System) then it is recommended that its format be similar to that of the Description Block and the Design Block.

Appendix K - U-0024

## 1. Introduction

The initial motivation for the project was the desire to improve the quality of R & D (UK) software output.

This was to be done by considering the developing of a 'Structured Programming System'. It was originally thought that this would consist of:

software tools
programmers guide (including standards)

Because this is being considered by PEOPLE, for PEOPLE to USE the approach taken was to solicit peoples opinions. R & D (UK) staff were asked to consider the following topics:

design techniques
design languages
program layout
documentation
structured coding methods
tools

This developed naturally to include a number of other areas of concern to the individuals polled.

The following sections represent as closely as possible the views expressed during the discussions held.

It is important to notice that an area of general concern was project organisation. People were interested in seeing the structured approach applied throughout a project.

The views expressed in this document will be used as input to the SPS project.

## 2. Design Techniques

1. Design techniques desparately needed.

2. People aware of (and attempted) a 'Top-Down' approach but some mutations ie 'Middle-Out'

3. Appreciation of value of obtaining a whole design before coding starts but not always adhered to

4. They felt that a comparison of currently known techniques would be useful but that a justified recommendation of a technique would be satisfactory.

5. A recommended methodology must be usable, in the normal working environment

6. Strong requirement for recommendations on project organisation

7. Techniques used should encompass and facilitate testing of the design eg walkthroughs

8. Project organisation should include definition of documents to be produced.

9. The design methodology adopted should encourage consideration of the future testing requirements

3. Design Languages (DL)

1. People felt that these can be useful but must be usable

2. People felt that self-discipline is needed to maintain a design language problem statement in an up-to-date form. Many people felt reluctant to trust a design language statement of a problem for this reason.

3. To encourage widespread use of a DL the language chosen must be liked and easy to maintain

4. Finite State Diagrams have been adapted to this area for problems involving critical manipulation of variables and events.

5. Flowcharts are sometimes used but people felt constrained by the technique. Designs were not taken to a detailed level with this approach.

6. Michael Jackson Technique as a new approach has been tried and found to have deficiencies. The philosophy is accepted and found useful but its representation is difficult to handle and/or maintain.

7. Warnier Diagrams as a new approach has been tried and is gaining in popularity.

8. R-notation was felt to be designed for use with assembly languages. It was felt by some people to be superfluous in conjunction with a high- level language eg PL/1.
Those people supposedly using it have extended it.

9. An evolution from R-Notation is the development of structured commenting.

10. People would like to see a mapping between DL constructs and

programming language constructs for commonly used programming languages.

11. From 10 the problem of conflict between efficiency and style arises and guidelines are required.

## 4. Program Layout

1. People were familiar with the mandatory requirement for a '3-line header' and accepted this.

2. Few people were familiar with the extension to this proposed in PE-A-49.

3. The benefit of keeping routines to 2 listing pages or less was appreciated by a majority of people, but concern was expressed over the effect of incorporating large comments into these routines.

4. Though people were agreed on the benefits of commenting their code, this was sometimes done after the code was felt to be correct.

5. If a standard layout was adopted some interest was shown in a process where retrospective checking of this was performed.

6. Doubt was expressed as to the value of including distributed comments in a 'short' routine written in a high-level language.

7. It was felt that if a standard block comment was introduced it should include more than that proposed in PE-A-49
   eg information on external program entities
      revision numbers

8. The current conflicts between 80 column and 120 column media causes some problems.

9. People see a need for simple layout controls
   eg form feeds

## 5. Documentation

1. General problems in this area due to lack of direction and standards. All understood the requirement for documentation to be produced.

2. People felt that they would benefit from the existence of guidelines as to what documents should be produced during a project.

The guidelines should contain templates and/or checklists with respect to the minimum contents and structure of each document.

3. Document naming conventions should exist, allowing documents to be easily identified.

4. People proposed a number of possible documents and these proposals have contributed towards the recommended documents.

## 6. Structured Coding Methods

1. People expressed doubts as to the usefulness of structured coding methods applied to unstructured programming languages.

2. People would like to see some DOs and DONTs for each commonly used language, in terms of efficiency and style.

3. People felt that the use of Structured Coding Methods should not lead to the generation of multitudes of small routines (for its own sake) without regard for efficiency.

4. People have used various naming conventions to distribute information through their sources. Some people thought that there was a benefit to be obtained from this being formalised.

## 7. Tools

1. The only existing tool that people seemed interested in discussing was the indenter.

   People are generally dissatisfied with the formatting performed by the indenter, but some are prepared to use it.

2. A tool suggested was one that would check for adherance to layout standards.

3. People have an awareness of the possibility of extracting comments from programs.

   A desire to have this output compatible with their documentation was expressed.

4. The indenter
   - felt to be necessary to improve readability
   - differing views expressed on formatting required
   - felt that good layouts are destroyed

5. If a high-level design technique is to be used that involves some sort of schematic representation then tools must be made available to handle this.

## 8. Organisation

This section is included in response to peoples requirements.

1. There is an interest in the organization of project ufds to assist in project administration. These could be sub-divided as follows:

   sources
   binaries
   documentation
   etc

2. Interest was expressed in the creation of various library structures for use by R & D (UK) personnel. If these are created an administrative mechanism/tool must be available and be used.

   - utilities
     generally useful pieces of software that do not belong in CMDNCO
     information on these must be maintained
   - source subroutines
     generally useful source subroutines not appropriate for inclusion in APPLIB
     these should be source loaded into peoples programs, and must not include any insertions that are non-standard
     these must all include a standard block comment
   - standard declarations
     people would like to see declarations available for standard library subroutines
     these should be source loaded into peoples programs

3. Attention was drawn to the Proposed Source File System. This may impact some or all of the proposed tools

REFORM: Design Formatter [Rev 2.0]

The correct command line format is:
REFORM input_treename [output_treename]
plus the optional keywords:

| | | |
|---|---|---|
| -NQ | or -NO_QUERY | to allow overwriting permission |
| -NUC | or -NO_UPPER_CASE | to prevent converting keywords to upper case |
| -UCL | or -UPPER_CASE_LABELS | to force labels to upper case |

INFORM: INstruction FORMatter [Rev 2.0]

The correct command line format is :
 INFORM <input> [<output>][<option parameter>]
 The option parameters are :

| | | | |
|---|---|---|---|
| -NQ | or | -NO_QUERY | to allow overwritting permission |
| -CCOL xx | or | -COMMENT_COL xx | |
| -LM xx | or | -LMARGIN xx | column from which identation is measured |
| -RM xx | or | -RMARGIN xx | |
| -IND xx | or | -INDENT xx | spaces for each level of identation |
| -FILL | | | default |
| -NF | or | -NO_FILL | |

( xx = decimal number )

RESTATE: REpreSenTATion convErter [Rev 2.0]

The correct command line format is:
RESTATE input_treename [output_treename]
plus the optional keywords:

-NQ or -NO_QUERY to allow overwriting of output file

-xxx to indicate the required format of the output file when no output file is specified, and the format is to change
xxx must be one of:
PL1, PLP, PL1G, FTN, F77, PMA, DES, PASCAL, COBOL, BASIC, CPL, LISP, EMACS
when this option is used the output file name is constructed from the body of the input_treename and xxx

DENOTE: DEsign NOTEbook builder [Rev 2.0]

The correct command line format is:
DENOTE [input_treename] [output_treename]
plus the optional keywords:

| | |
|---|---|
| -CAT or -CATALOGUE | to treat the input as a list of files |
| -DGN or -DESIGN | to extract only design blocks |
| -DESC or -DESCRIPTION | to extract only description blocks |
| -ADJ or -ADJUST | to generate output in RUNOFF adjust mode |
| -NQ or -NO_QUERY | to allow overwriting of output file |
| -WID x or -WIDTH x | to set runoff line width to 'x' chars. |
| -BL x or -BLANK x | to set runoff space character to 'x' |
| -IS or -INFORM_SPLIT | to inform user of split lines |
| -NM or -NO_MESSAGE | to put all error messages into a file |
| -R or -REPORT | to report program statistics |

TEMPLATE: File Construction Utility [Rev 2.0]

The correct command line format is:

```
TEMPLATE <name>[.<suffix>] [options]
options  -PATH <pathname>
         -NO_QUERY or -NQ
         -<suffix>
```

ACCEPTABLE SUFFIXES

| LANGUAGE | SUFFIX |
|---|---|
| PL/I | - PL1 or PL1G or PLP |
| Fortran | - FTN or F77 |
| Prime Macro Assembler | - PMA |
| Cobol | - COBOL |
| Pascal | - PASCAL |
| Basic | - BASIC |
| Command Procedure Language | - CPL |
| Lisp | - LISP |
| Emacs | - EMACS |
| Design | - DES ( NOTE: this will not give a code block ) |

* NOTE: All products have been modified to conform to master disk standards. For a description of these modifications, please read INFO19>STANDARDS.RUNO.