SG/4.2

Starlink Project
Starlink Guide 4.2

M D Lawden, K F Hartley

12 August 1992

ADAM — The Starlink Software Environment

Abstract

This is a comprehensive description of the Starlink Software Environment as of 1992.

Contents

List of Figures

Part I

Preface

Acknowledgements

The initial revision of the original version of this document was carried out by Ken Hartley. It was then taken over by Mike Lawden who put it into its final form.

The source documents that have been used are extensively referenced within the text. We would like to thank the authors of these documents for making them available. We would also like to thank the many people who made comments on the working drafts of this guide.

Preface

Whoever has lived long enough to find out what life is, knows how deep a debt of gratitude we owe to ADAM. **Mark Twain**

ADAM

ADAM — the acronym originally stood for the Astronomical Data Acquisition Monitor — had its beginnings in the Royal Observatories and is now the cornerstone of software support for the UK astronomical community. This document focuses on its role within Starlink for data-reduction and analysis, for which it is well suited. However, it should not be forgotten that it is perhaps even more important for data acquisition at the UK's overseas observatories, where it has no competitors.

ADAM has three major components — applications; subroutine libraries to help build applications; and command languages to run applications. The libraries can be thought of as a toolkit, or a collection of building blocks for constructing applications. *Lego,* the children's toy, provides a good model — because care was taken to define the interface properly (size and spacing of knobs and holes) it is guaranteed that any piece will fit onto any other piece, even when that connection was not envisaged when the original piece was designed.

The command language forms the working environment of the user, and the libraries themselves are sometimes seen as the software environment in which programmers work. However, these are both subsidiary to the applications which are available to help astronomers to carry out their research.

History

Within a year of Starlink being set up, the *Interim Starlink Environment* was delivered and working. It was always recognized that the facilities it offered were limited — it consisted of only a handful of routines to handle parameters and a simple image format. The DSCL command language was soon added.

Work started at once on the definitive Starlink Software Environment (SSE). For various reasons — insufficient manpower and the untimely death of its chief architect being the main ones — a satisfactory product was never delivered. In the meantime, the author of DSCL had moved into the La Palma software team at RGO and built ADAM to control instrumentation for the INT and JKT on La Palma, using Perkin-Elmer computers. ROE then adopted ADAM for instrument control on UKIRT and ported it to VAX/VMS. It used the Hierarchical Data System and SGS/GKS graphics packages developed at RAL as part of the SSE project. ROE also re-implemented the SSE parameter system. Though closely compatible with the SSE, the new version performed much better, in part because character handling was made more efficient (but at some cost in future compatibility with Unix and C).

By 1985 it was clear that Starlink was unlikely to have the resources needed to provide what the community required. The Project therefore recommended adoption of the IRAF system from the USA. Workshops were held in late 1985 and early 1986, which were attended by astronomers representing all major groups of Starlink users. After a thorough review of many systems (including IRAF, MIDAS, AIPS, variants of the Interim Environment, IDL, LUCID, ADAM and many more) the community decided that ADAM offered the best overall prospects. It was proposed that further development and support would be on a ‘best endeavours’ basis by all the interested parties. These proposals were endorsed by the Starlink User Committee.

Subsequently the VAX version of ADAM was adopted as the basis of instrument control at AAT, for the WHT on La Palma, and the JCMT on Mauna Kea. In 1989 ICL was adopted as the command language for ADAM, replacing the earlier ADAMCL. The need for a more formal method of supporting ADAM was recognized and so in 1990 the ADAM Support Group was established within the Starlink project at RAL.

It is now recognized that the cost/performance ratio of RISC (Reduced Instruction Set Chip) machines means that they currently provide typically three times the performance of VMS systems for the same price. This has inevitably led to growing numbers of astronomers using workstations — particularly Suns and DECstations — running varieties of the Unix Operating System. In order to ensure that the investment in Starlink application software is not wasted, and the community continues to have the enviable coherence which Starlink provides, the decision was taken in 1991 to remove all the dependence on VMS from ADAM. The result is a *portable* version of ADAM. This version has proved to perform very well on Unix systems.

The importance of ADAM

The above history shows that ADAM is essential for every major telescope which is funded through SERC. Its role in Starlink means that all UK observational astronomy depends on ADAM. The spread of Unix to big machines (Cray, IBM, massively parallel systems and so on) allied with a portable system means that even the computational modellers in the community can contemplate using ADAM, especially when high bandwidth communications make it feasible to distribute work between a local workstation and a remote supercomputer.

ADAM was chosen by the community and is now supported by SERC. It was chosen because it was the best possible option at that time. It is fully under the control of the UK community and is not dependent on any one individual. It has been designed to handle any astronomical data and is guaranteed to be supported on all Starlink hardware, and graphics devices in particular. The existence of a portable version means that it can be made available on almost any hardware platform.

Chapter 4 describes many packages built using ADAM, ranging from large, general-purpose packages such as FIGARO and KAPPA, through those aimed at specific branches of astronomy (*e.g.* ASTERIX, TSP), to those more concerned with software and data handling (*e.g.* SST, CONVERT). New astronomical applications are being added all the time. It is hoped that this Guide will lead more people to use them and encourage users to write their own.

Part II

INTRODUCTION

Chapter 1
Introduction

1.1 Who is this document for?

This Guide is for anyone interested in Starlink software. Its purpose is to provide people with a single document that describes the ADAM software environment in sufficient detail for them to understand what it is, what it can do, and where to find more information. It is only for people doing data analysis. Features of ADAM which are particularly relevant to data acquisition are not covered.

1.2 What is ADAM?

This question is often asked, and various equally valid answers are possible. This Guide is based on the view that ADAM is about *astronomical applications software* written in a particular way. Specifically, it is about applications written using the *ADAM subroutine libraries*. It has usually been found that when a collection of application programs has been developed, the need is perceived for a *command language* to invoke them and to offer useful features over and above those provided by the operating system command language. These, then, are the three major components of ADAM: **Applications, Subroutine libraries, Command language(s)**. However, there are other aspects to ADAM.

Firstly, it is about a certain style in constructing applications, sometimes called a software *architecture*, and about coding *standards*. ADAM also stands for *professional* software of high quality. It means that attention is given to *supportability*, *portability*, and *performance*.

Secondly, it is concerned with a consistent way of handling the storage of large quantities of *astronomical data* from all possible sources.

Many ADAM libraries can be used in isolation to write 'non-ADAM' applications. This practice is encouraged as it makes it more likely that such applications will fit in better with existing ADAM applications.

It must be emphasized that this document refers to the VMS version of the software being discussed. There will be minor differences when ADAM is ported to other systems. Subsequent versions of this document will specify those differences.

1.3 The structure of this document

Part I presents the fundamental information you need to know before you can use ADAM successfully. It also give you a guided tour in which ADAM is used to process simple data in simple ways. The idea is to give you a feel for what the system is like, without drowning in details.

Part II begins by surveying the applications software that is available for use. It then describes in detail the main language that is used to run ADAM programs, namely ICL. It ends by introducing the data system (HDS/NDF) that is at the heart of ADAM.

Part III shows how to use ADAM subroutine libraries to create new applications. It begins by describing some simple ADAM programs, and how to compile, link, and run them. ADAM programs are associated with something called an Interface File which describes the parameters which control the program — this is also described. Another important consideration for ADAM programmers are the standards and conventions which are recommended when writing programs, and the tools which are available to help you do it, and a chapter is devoted to this. Next, the ADAM libraries are surveyed, and then the individual systems are described in more detail in separate chapters.

Part IV is a reference section containing a summary of the main command language, ICL, and lists of the precise contents of ADAM application packages and subroutine libraries. The number of individual programs and subroutines is so large (about 1500 of each) that it is helpful to have them organised in functional groups and listed concisely in a consistent format without (in the case of subroutines) the clutter of parameter lists. This should enable you to understand in detail just what the packages and libraries can do. When you start to use the subroutines in your programs, you will need their associated documentation in order to establish the required parameters.

Chapter 2
Getting Started

This chapter gets you started with ADAM. If possible, you should try out the examples on your terminal so you can verify that what is claimed to happen does happen, and you can get a feel for the ADAM system. Any terminal will do — you don't need image displays or graphics devices.

If you are totally unfamiliar with computers you should read the introductory guide to your local computer system (for example, *Introduction to VMS*), or better still get the Starlink site manager or a local user to show you. In the rest of this manual you are assumed to be familiar with basic VMS terms and concepts.

2.1 Preliminaries

Before you switch on your terminal, let us explain how to interpret the examples. The basic principle we use is to show lines as they actually appear on your terminal screen. When the computer wants you to type something it normally displays a prompt. For example, the VAX VMS operating system displays the prompt:

```
    $
```

(We indent lines which appear on your terminal screen.) You respond to this prompt by typing something that VMS will understand. For example:

```
    $ DIR
```

will cause the contents of your default directory to be displayed on the screen. What you actually type are the characters 'DIR', followed by the 'carriage return' key[1]. We will use the word 'enter' to mean 'type and then press carriage return'. A displayed line will consist partly of characters output by the computer, and partly of characters typed in by you. We do not attempt to differentiate between the two by typographical conventions; you will find it obvious which is which when you try out the examples. Sometimes the only key you need to press is the 'carriage return' key. Usually, this means that you want to accept a default value offered by the parameter system. For example, you might come across a line like:

```
    XDIM - x dimension of output array /64/ >
```

This is output by the computer, and the last character '>' is an invitation for you to type something. If the line appears like that without any further comment, it means that you respond by pressing 'carriage return'. Finally, you will find that in most cases you can type input characters in upper or lower case — it doesn't matter which; they are interpreted as being the same.

Sometimes you need to press a key while holding down the 'CTRL' key. This is the way you send 'control codes' to the computer. This is indicated typographically by the convention 'ctrl/x', where 'x' is the other key you press. For example, 'ctrl/Z' means 'press the Z key while holding down the CTRL key'.

[1]Sorry, but there are still people around who complain that you never told them this.

2.2 Quotas

You must have sufficient VMS process quotas for running ADAM. Typical quotas required are as follows:

CPU limit:	Infinite	Direct I/O limit:	18
Buffered I/O byte count quota:	65400	Buffered I/O limit:	18
Timer queue entry quota:	10	Open file quota:	74
Paging file quota:	49000	Subprocess quota:	10
Default page fault cluster:	64	AST quota:	23
Enqueue quota:	100	Shared file limit:	0
Max detached processes:	0	Max active jobs:	0
JTQUOTA:	3072		

The paging file quota given here is large enough to allow three large applications to be loaded simultaneously. You can find out what your quotas are by:

```
$ SHOW PROC/QUOTA
```

You may run out of paging file quota when using big programs or data sets. This is usually indicated by the system failing to do what you asked it to do and displaying some (probably incomprehensible) message on the screen. Consult your Site Manager if you need to get your quotas increased.

2.3 Starting up ADAM

If you have not already executed the Starlink command procedure SSC:LOGIN.COM (which defines symbols like ADAMSTART and ICL), do that first:

```
$ @SSC:LOGIN
```

Since you need to do this before using any Starlink software, it is worth putting it in your personal LOGIN.COM file.

The first thing to do when you want to use ADAM is to type in the following command:

```
$ ADAMSTART
*****************************************
* ADAM Version 2.0-1 has been installed *
*  Type ADAM_CHANGES for information    *
*****************************************
  ADAM version 2.0-1 available
$
```

(If this command causes problems, then ADAM hasn't been installed properly at your site — see your Site Manager. You may, of course, find that you are using a different version of ADAM, so you may get a slightly different response when you try it.) This command sets up various logical names and commands needed to use ADAM. Also, if you haven't executed this command before, it will create a subdirectory called ADAM in your login directory and give it the logical name ADAM_USER[2]. You can see what's in it by typing:

```
    $ DIR ADAM_USER
```

Notice that it contains a file called GLOBAL.SDF; this will hold global parameter values. (In general, the behaviour of programs and the files they read and write can be modified by specifying parameters, of which much more will be said later. Global parameters are parameters which are not specific to any one application.) The file type qualifier '.SDF' is characteristic of ADAM data files. They are written by a component of ADAM called HDS, which stands for 'Hierarchical Data System'. ADAM_USER is used by the system to hold files which it creates automatically. It should be cleared occasionally by using the VMS PURGE and DELETE commands in the usual way. Provided that no parameter values need retaining between sessions, everything can be deleted at the end of a session, but ADAMSTART must be re-run to initialize things again before ICL is used.

2.4 Choice of command language

As an ADAM user you have a choice of at least three ways of running ADAM programs:

- directly from DCL (the 'VMS' command language)
- from ICL (the 'ADAM' command language)
- from SMS.

ICL is recommended for most purposes because it gives flexibility and good performance when you are using many commands. It also offers you an easy, yet efficient, way of writing simple procedures, and a comprehensive scientific calculator (with the ability to feed the results directly into programs). It also allows you to run multiple applications at the same time. Most of this Guide will assume that ICL is being used.

DCL gives fewer facilities — particularly in terms of command line specification — but in some circumstances it can be quicker. Naturally, it also gives immediate access to all DCL facilities. For this reason it is often quickest to test a new program under DCL; the normal switching between editor, compiler and program execution could mean frequent overheads in either stopping and starting ICL or having to execute DCL commands from ICL.

SMS ('Simple Menu System') is primarily used for data-acquisition, though there is no reason why it should not be used for data-analysis. Its main advantage is that it requires little typing, and gives maximum efficiency when a few programs are being run which have lots of parameters but only a few change for each run. Its main drawbacks are that it is quite difficult to set up the description of what is to appear in the various menus and how they are to look, and that this has not been done for the normal Starlink application packages.

SMS is not considered further in this guide. If you wish to investigate it, read chapter 11 of the ICL User's Guide (SG/5).

[2] ADAMSTART will only create the [.ADAM] directory and define the *job* logical name ADAM_USER if ADAM_USER is not already defined as a *job* logical name. If ADAM_USER is so defined, the assigned directory is assumed to exist.

2.5 Trying out the command languages

The program TRACE, used below, looks at the contents of a data file of the type handled by the Hierarchical Data System (HDS).

2.5.1 ICL

You start up the ICL command language by typing:

```
$ ICL
```

This loads the ICL program which will interpret the commands you type. After it has warmed up (a few seconds), it displays some introductory information:

```
Interactive Command Language   ---   Version 1.5-7

   - Type HELP package_name for help on specific Starlink packages
   -   or HELP PACKAGES for a list of all Starlink packages
   - Type HELP [command] for help on ICL and its commands

ICL>
```

You may find that the system responds differently when *you* try it because it may have been updated since we wrote this. Anyway, it's all self-explanatory. The last line is the prompt that ICL gives when it wants you to enter something. Like the '$' prompt, the '`ICL>`' prompt is output by the computer; you do not type it in yourself.

You can find out what packages are available at your site by entering:

```
ICL> HELP PACKAGES

PACKAGES

    The following ADAM applications packages are available from Starlink:

    Standard Packages:
     CONVERT   -  Data format conversion.
     FIGARO    -  General data-reduction.
     KAPPA     -  Image processing.
     SPECDRE   -  Spectroscopy Data Reduction
     SST       -  Simple Software Tools
     TSP       -  Time series and polarimetry data analysis.

    Option Packages:
     ASTERIX   -  X-ray data analysis.
     CCDPACK   -  CCD data reduction
     DAOPHOT   -  Stellar photometry.
     PHOTOM    -  Aperture photometry.
     PISA      -  Position, Intensity and Shape Analysis
     SCAR      -  Catalogue access and reporting.

    Additional information available:

    ASTERIX    CCDPACK   CONVERT    DAOPHOT    FIGARO     KAPPA
```

```
        MISCELLANEOUS        PHOTOM     PISA       SCAR       SPECDRE   SST
        TSP

    PACKAGES Subtopic?
```

Now you can enter the name of one of the packages to get information on it. Use it exactly like the VMS HELP system. Keep pressing 'carriage return' to get back to the 'ICL>' prompt.

One of the things you can do with ICL is get answers to calculations; for example, if you input:

```
    ICL> X=SQRT(2)
    ICL> PRINT Result=(X)
```

ICL will print the answer:

```
    Result= 1.414214
    ICL>
```

You can exit from ICL at any time by entering the command:

```
    ICL> EXIT
    $
```

The '$' prompt indicates that you are now talking to DCL. To use ICL again you will have to type:

```
    $ ICL
```

again, and this will generate the same introductory lines as before.

When you have an ICL> prompt, simply type:

```
   ICL> TRACE
```

followed by <CR> when you get the prompt:

```
   OBJECT --- Object to be examined /@ADAM_USER:GLOBAL/ >
```

This will show you the structure and contents of the ADAM_USER:GLOBAL.SDF file.

Once again, to leave ICL and return to DCL simply type:

```
   ICL> EXIT
   $
```

The above sequence of operations is encapsulated in the single command `ADAM`. Therefore, the easiest way of running `TRACE` is to type:

```
    $ ADAM TRACE
```

which leaves you in ICL and in a position to try out any of the other ICL commands.

2.5.2 DCL

When using DCL, the normal way to run program TRACE is simply to type:

```
   $ TRACE
```

and then press <CR> when you receive the prompt:

```
   OBJECT --- Object to be examined /@ADAM_USER:GLOBAL/ >
```

Once again, the contents and structure of the file will be displayed, only this time you will be left in DCL with a '$' prompt.

Chapter 3
A Guided Tour

This Chapter contains simple illustrations of how to use ADAM applications from the ICL command language.

One of the best ways of learning how to use ADAM is to play with one of the applications packages. There are many of these and they are described in Chapter 4. The KAPPA package is used below because it conforms to ADAM conventions and has a lot of useful commands for processing data.

Remember, before you attempt to use ADAM applications from ICL you must initialise ADAM logical names:

```
$ ADAMSTART
```

and start up the ICL language interpreter:

```
$ ICL
```

3.1 Starting KAPPA

KAPPA is made available from within ICL by typing:

```
ICL> KAPPA
```

This loads KAPPA and defines the commands required to run its programs. It also generates the following output:

```
 Help key KAPPA redefined

 --     Initialised for KAPPA     --
 --    Version 0.8, 1991 August   --

 Type HELP KAPPA or KAPHELP for KAPPA help

ICL>
```

Don't worry about these messages for the moment. If you now enter:

```
ICL> HELP KAPPA
```

you will get an introduction to the KAPPA Help system:

```
HELP

   This is the KAPPA online help system.  It invokes the VMS HELP Facility
   to display information about a KAPPA command or topic.  If you need
   assistance using this help library, enter "Using_help" in response to
   the "Topic?" prompt.  If you need more information about getting help
   about KAPPA from the ICL level, then enter "Command-line_help".
```

followed by the prompt:

```
Topic?
```

This is an invitation for you to ask for further information. To find out what subjects are available you simply enter a '?'. You may find the following topics particularly useful :

Classified_commands — displays a list of subject areas as subtopics. Each subtopic lists all KAPPA applications in that classification, and gives their functions.

Summary — gives a brief description of the function of each application within KAPPA.

Using_Help — tells you how to use this particular Help system.

<**command**> **param** — describes all the parameters of the particular KAPPA `<command>`. Subtopics give further details about individual parameters, including their command line position.

You can either enter a subtopic, or press <CR> to get back the 'Topic?' prompt. Then you can enter '?' to get the list of topics displayed again, or just press <CR> to get back to the 'ICL>' prompt. You can escape from the Help system at any level by entering ctrl/Z.

Now have a look in your ADAM_USER directory:

```
ICL> DIR ADAM_USER
PROCERR    Procedure DIR not recognized
ICL>
```

The attempt to use the DIR command fails[1] because it is only recognized by the DCL command interpreter, and you are talking to the ICL command interpreter. What you need to do is to tell ICL that this is a DCL command. You can do this by preceding the command with a '$' character, thus:

```
ICL> $ DIR ADAM_USER
```

ADAM now creates a subprocess in which to execute the DCL command processor. This is the reason for the delay and the message:

```
Creating DCL subprocess
```

This subprocess is only created once per session; any further DCL commands you issue will be run in the DCL subprocess without any further action from you.

3.2 Creating test images

You can now use KAPPA to create some small images which you can then display and manipulate. The simplest way to create an image is to use command CREFRAME, which you can type in lower case if you like:

```
ICL> creframe
```

[1]It is intended to change this soon.

After a loading message, the first thing that happens is that you are invited to specify a value for one of the program parameters:

```
    XDIM - x dimension of output array /64/ >
```

(If you have used CREFRAME before, you might find the '/64/' in the above line comes out as something else.) This line is typical of the way in which ADAM asks for *parameter values.* The first field, in this case 'XDIM', is the parameter keyword — the name by which the parameter is known to the user. Then, after the '-' delimiter, comes the second field, in this case 'x dimension of output array'. This is a brief description of what the parameter means. The third field, in this case '/64/', shows a suggested value for the parameter. If you just press <CR> in response to the prompt, the suggested value will be taken as the value you wish to specify for that parameter. Sometimes a suggested value will not be displayed in a prompt. In this case you *must* specify a value.

For the purposes of this example it is sensible to choose a small array size, say 10×10. Thus, the suggested value of '64' is not what you want and you must specify the value you require by entering the value '10':

```
    XDIM - x dimension of output array /64/ > 10
```

The parameter system then accepts this value as the value of the parameter XDIM. Once you press <CR>, the system will present you with a prompt for another parameter value:

```
    YDIM - y dimension of output array /64/ >
```

This prompt is similar to the previous one, and the suggested value is '64' as before. Specify the value '10' again:

```
    YDIM - y dimension of output array /64/ > 10
```

The system now presents you with the following lines:

```
    GS = Gaussians, RR = Random 0 to 1, RL = Random from Min to Max,
    RA = Ramp across image, FL = Flat, BL = Blank,
    GN = Gaussian noise with standard deviation about the mean,
    RP = Poissonian noise about mean.
    TYPED - Type of data to be generated /'GS'/ >
```

The first four indicate the codes you can enter in response to the parameter prompt and the meanings attached to them. The last is a prompt in the same format as the previous prompts with a suggested value of 'GS'. In this case, the input required is a character string rather than an integer — you can usually tell what type of input is required from the format of the suggested value. For this illustration, generate an array that is simple in structure to make it easy to follow what is going on. Choose an image in the form of a 'ramp' — this is an array whose pixel values rise or fall steadily from left to right or from top to bottom. The code is 'RA':

```
    TYPED - Type of data to be generated /'GS'/ > ra
```

You may wonder why the suggested value is surrounded by quotes but the supplied value is not. In fact, the supplied value should strictly be `'RA'`, but it is permissible to omit the quote characters as long as there is no ambiguity in doing so[2].

The next line to appear is:

[2]This is not the right place to become embroiled in a discussion of what is a quite complicated subject — see Chapter 8.

```
    LOW - Lower limit for data >
```

This time the suggested value field (the '/.../' bit) is missing. Whether it is present or not depends on how the programmer has used the parameter system and whether or not he thinks a suggested value is appropriate. Because no suggested value is given, you need to enter a value — use '1' for the lower limit and '10' for the upper limit:

```
    LOW - Lower limit for data > 1
    HIGH - Upper limit for data > 10
```

The next prompt is:

```
    DIRN - Direction of ramping /1/ >
```

This time you have suggested value again, but what does the number mean? You can find out by entering '?':

```
    DIRN - Direction of ramping /1/ > ?
```

The system will explain that '1' generates pixels which increase from left to right, '2' generates pixels which increase from right to left, '3' bottom to top, and '4' top to bottom. Choose the suggested value '1' by just pressing <CR>:

```
    DIRN - Direction of ramping /1/ >
```

The next prompt to appear is:

```
    OUTPIC - Image for output data >
```

Here there is no suggested value, so you must enter your own. The response required is a name for the generated image; call it RAMP1:

```
    OUTPIC - Image for output data > ramp1
    ICL>
```

After 'ramp1' is entered the program generates the required image, so there may be a slight delay. The 'ICL>' prompt indicates that the CREFRAME command has completed its task and has returned control to the command language processor. The generated image will be called 'RAMP1' and will be stored in a file called RAMP1.SDF in your default directory — take a look, you will find it is there.

If you now generate three more 10×10 images containing different types of ramp, you can display and process them in various ways. Each time you want to create a new image using CREFRAME, you must enter the name of the program as a command:

```
    ICL> creframe
    XDIM - x dimension of output array /10/ >
```

Notice that the second and all subsequent times you run an application program, there is no delay while it is loaded; the response is almost immediate. This is one of the main advantages of running applications from ICL. Notice also that the suggested value has changed from '64' to '10' — the parameter system has remembered the value you last entered for XDIM; this is called its 'current value'. (The current values of parameters used by KAPPA commands are stored in the file KAPPA.SDF in your ADAM_USER directory.) Thus, you can just press <CR> to accept the suggested value. Generating the next image is going to be easy, isn't it?

```
YDIM - y dimension of output array /10/ >
GS = Gaussians, RR = Random 0 to 1, RL = Random from Min to Max,
RA = Ramp across image, FL = Flat, BL = Blank,
GN = Gaussian noise with standard deviation about mean,
RP = Poissonian noise about mean.
TYPED - Type of data to be generated /'GS'/ >
```

Wait a minute — last time you specified 'RA' as the value of TYPED, but the suggested value has remained 'GS' as it was originally. In this case, the suggested value 'GS' is generated by the program and the current value is not used. Once again, it all depends on how the parameter system has been used by the programmer — this subject will be considered in more detail in Chapter 14. All you can do is keep a careful eye on the suggested values. Here you must specify 'RA' explicitly again in order to generate a ramp:

```
TYPED - Type of data to be generated /'GS'/ > ra
LOW - Lower limit for data > 1
HIGH - Upper limit for data > 10
DIRN - Direction of ramping /1/ > 2
OUTPIC - Image for output data > ramp2
ICL>
```

This time DIRN has been set to '2' and the image called 'RAMP2'.

In a similar way, you can now generate images called 'RAMP3' and 'RAMP4' with DIRN set to 3 and 4 respectively. Notice that the suggested value for DIRN is similar to that for TYPED in that it always has the same value rather than the last used value.

3.3 Examining images

Now that you have generated four images, you can play with them. In order to avoid getting involved with graphics devices at this stage, only commands which will run on any terminal will be used. The first thing to do is to examine their pixel values. There are several ways of doing this with KAPPA; one way is to use the LOOK command:

```
ICL> look
INPIC - Image to be inspected/@ramp4/ >
```

The suggested value for the name of the image to be looked at is '`@ramp4`'. The '`@`' character means 'an object with the name of', thus '`@ramp4`' means 'an object with the name of ramp4'. In this case the system assumes that any character string you type is the name of a data object; therefore you do not need to precede the name with an '`@`' symbol. In the example, the system realizes that the value of INPIC should be the name of an image and it remembers that the name of the last image referred to was 'ramp4'. However, look at 'ramp1' first:

```
INPIC - Image to be inspected/@ramp4/ > ramp1
```

LOOK now accesses 'ramp1' and displays the following information:

```
Title = KAPPA - Creframe
Array is 10 by 10 pixels
```

The title of the image — 'KAPPA - Creframe' — was generated by the CREFRAME program, and the array size is 10×10 as expected. This is followed by the next prompt:

```
CHOICE - Peep, Examine or List /'Peep'/ >
```

These are just different ways of listing the pixel values. The suggested value is 'Peep', so try that by typing <CR>. You can also accept the suggested values for the next two prompts which specify the central point of the part of the image you are going to peep at:

```
XCEN - x centre pixel index of 7x7 box /5/ >
YCEN - y centre pixel index of 7x7 box /5/ >
```

The 'Peep' option will display the pixel values within a 7×7 box centred on pixel (XCEN, YCEN). Notice that the program has chosen the centre of the image as the suggested box centre. All the required parameter values have now been entered, so the program proceeds to display the specified box:

```
      2         3         4         5         6         7         8

8     2         3         4         5         6         7         8
7     2         3         4         5         6         7         8
6     2         3         4         5         6         7         8
5     2         3         4         5         6         7         8
4     2         3         4         5         6         7         8
3     2         3         4         5         6         7         8
2     2         3         4         5         6         7         8

ANOTHER - Another inspection ? /YES/ >
```

Pixel indices are numbered from bottom left. In this case the pixel values are equal to their column number. Unlike CREFRAME, you are not returned to ICL but are given a chance to peep at another part of the image — these decisions are made by the programmer. As the image you are peeping at is highly predictable, do not bother to examine it further but escape from the program by entering:

```
ANOTHER - Another inspection ? /YES/ > n
ICL >
```

When a program asks for a 'YES' or 'NO' value, you can get away with 'Y', 'True', and 'T' for 'YES'; and 'N', 'False', and 'F' for 'NO'; regardless of case.

You can use the LOOK command to examine the pixel values of the other images you have created in a similar way to that shown above. The 'List' option gives you the opportunity to store the output in a file:

```
ICL> look
INPIC - Image to be inspected/@ramp1/ >
Title = KAPPA - Creframe
Array is 10 by 10 pixels

CHOICE - Peep, Examine or List /'PEEP'/ > L

XLOW - x start pixel index of sub-array /1/ >
YLOW - y start pixel index of sub-array /1/ >
XSIZE - x size of sub-array /10/ >
YSIZE - y size of sub-array /10/ >
FILENAME - Name of listing file /@LOOKOUT.LIS/ >
Listing to LOOKOUT.LIS

ANOTHER - Another inspection ? /YES/ > n
ICL>
```

Notice in this case that the parameter prompts after that for 'CHOICE' differ from those encountered before. This is because you have chosen the 'List' option instead of the 'Peep' option, and the program needs different information before it can proceed. Accept the suggested values for all the parameters until you come to 'ANOTHER' again. This makes things quick and easy for you. The program writes the pixel listing to the file LOOKOUT.LIS which is written in your default directory. You should, therefore, be able to display it on the screen using the DCL 'TYPE' command.

```
ICL> $ type lookout
```

will cause the contents of file LOOKOUT.LIS to be displayed on your terminal (in a folded form — not shown here).

3.4 ICL procedures

So far we have used ICL to do simple things like calculate the value of a simple formula and load and use KAPPA. However, ICL can also be used as a programming language in which to write procedures, just as commands of DCL can be stored in a command procedure. This is a very convenient way to extend the range of applications available to you.

Suppose you want to add together (pixel by pixel) the four images you have created to produce a new composite image. This is the sort of problem that can be solved using an ICL *procedure*. ICL has a lot of functions available, one of which, 'SNAME', generates sequential names:

```
SNAME(string,n,m)
```

produces a name which is the concatenation of the 'string' with the integer n. A third parameter m specifies a minimum number of digits for the numeric part of the name. Leading zeros are inserted if necessary to produce at least m digits, thus:

```
SNAME('RUN',3,1)      has the value   RUN3
SNAME('IPCS',17,3)    has the value   IPCS017
```

An ICL procedure can be written using this to add a whole series of images together:

```
ICL> PROC RAMPADD N
RAMPADD> { Add images RAMP1 to RAMPN to form SUM
RAMPADD>   ADD RAMP1 RAMP2 SUM TITLE='Sum of 2 images'
RAMPADD>   LOOP FOR I=3 TO (N)
RAMPADD>     TITLE = 'Sum of' & I:2 & ' images'
RAMPADD>     ADD SUM (SNAME('RAMP',I,1)) SUM TITLE=(TITLE)
RAMPADD>   END LOOP
RAMPADD> END PROC
ICL>
```

The line starting `PROC` tells ICL you wish to write a procedure called `RAMPADD` which is to have one parameter called `N`. This will use the KAPPA application `ADD` to add successive images together in the new image `SUM`.

You can save this procedure for future use by entering:

```
ICL> save rampadd
```

To run this procedure, use its name as a command and specify the value of the parameter. Thus, to add together the images RAMP1, RAMP2, RAMP3, and RAMP4 enter:

```
ICL> rampadd 4
ICL>
```

An ADD statement is executed three times within the procedure, and a suitable title is generated for each image which is created. If you use LOOK to 'Peep' at the new image called 'SUM', you will see that every pixel has the value '22' as you would expect. Note that four SUM.SDF files have been created. You should PURGE or DELETE them.

End your ICL session by exiting from ICL in the normal way:

```
ICL> exit
$
```

You should now be fairly familiar with the style in which ADAM programs are used. More examples of how to use KAPPA are given in SUN/95. ICL procedures are considered in more detail in Chapter 9.

Part III

FOR USERS

Chapter 4
Applications Packages

When you run ICL and type:

```
    ICL> HELP PACKAGES
```

you will be shown two lists of packages. The first contains *Standard* packages which are available at all Starlink sites. The other contains *Options* which are made available at a site only on request.

This Chapter gives a brief overview of each package, while Chapter 20 describes the specific commands available within each package. Further information can be obtained from their associated Starlink User Note (SUN) which is referenced at the top of each section. Starlink software as a whole is described in SUN/1.

An important part of the rationalization of Starlink software which the coming of ADAM made possible concerns data structures. The Hierarchical Data System (HDS) (see Chapter 10) is very flexible, and is capable of creating an infinite variety of data structures. Without recommending some standard structure there would be a danger of programmers writing applications which could not read each other's data. If the standard is also implemented in a small number of routines, the restrictions imposed by the standard also make programming easier. The main unifying theme of Starlink applications is the standard data format defined by Starlink; this is the Extensible N-dimensional Data Format — NDF — described in Section 10.2. This is centred on an n-dimensional data array that can store most astronomical data such as spectra, images and spectral-line data cubes. The NDF may also contain such information as title, axis labels and units, and error and quality arrays. There is also a place to store ancillary data associated with the data array. These could be information about the original observing set-up, such as airmass during the observation or temperature of the detector; there may be calibration data or results produced during processing, for example spectral line fits. Groups of related parameters not defined by the NDF format itself are held in *extensions*.

A key component of Starlink software is KAPPA (Kernel Application Package). This does not process non-standard extensions, but neither does it lose them — it copies them to any NDFs which it creates. Other application packages may be able to process some, but not all, extensions. It is hoped that such packages will use KAPPA applications as templates, in procedures, or directly as appropriate.

Each section below contains a sketch of how to use the application being described. In these:

```
    $ command
```

means "issue the command from DCL", while:

```
    ICL> command
```

means issue it from ICL. Remember that before using any of these packages, the commands:

```
    $ @SSC:LOGIN
    $ ADAMSTART
```

should have been executed. Likewise, to run ICL you should type:

```
$ ICL
```

to get the ICL prompt.

The packages described in this chapter are listed below ordered by function. This should help you find a package which is appropriate for a particular purpose. The packages are then described in alphabetical order on separate pages.

Image Analysis & Photometry
- **KAPPA** — Kernel applications
- **DAOPHOT** — Stellar photometry
- **PHOTOM** — Aperture photometry
- **PISA** — Object finding and analysis

Spectroscopy
- **FIGARO** — General spectral reduction
- **SPECDRE** — Spectroscopy data reduction

Specific Wavelengths
- **ASTERIX** — X-ray data analysis

Specific Instruments
- **CCDPACK** — CCD data reduction
- **IRCAM** — Infrared camera data reduction

Polarimetry
- **TSP** — Time series and polarimetry analysis

Database Management
- **SCAR** — Catalogue data base system

Utilities
- **CONVERT** — Data format conversion
- **SST** — Simple software tools

4.1 ASTERIX — X-ray data analysis [SUN/98]

This is a collection of programs to analyse astronomical data in the X-ray waveband. Many of the programs are general purpose and are capable of analysing any data in the correct format. It is instrument independent and currently has interfaces to the Exosat and Rosat instruments.

ASTERIX data are stored in HDS files and are therefore compatible with all ADAM packages. There are basically two different types: binned and event datasets. *Binned data* (e.g. time series, spectra, images) are stored in files whose structure is based on the Starlink standard NDF format (SGP/38). Data errors (stored in the form of variances) and quality are catered for. *Event data sets* store information about a set of photon 'events'. Each event will have a set of properties, e.g. X position, Y position, time, raw pulse height.

The input data are first processed by an instrument interface. Event data are then processed and binned, and then the binned data are processed. Finally, graphical output is generated.

The commands may be classified as follows:

- Interface to a particular instrument (EXOSAT, ROSAT, etc).
- Event dataset and binned dataset processing.
- Data conversion and display.
- Mathematical manipulations.
- Time series analysis.
- Image processing.
- Spectral analysis.
- Statistical analysis.
- Data quality analysis.
- HDS editor.
- Source searching.
- Graphical and textual display.

To run ASTERIX:

from ICL:

```
$ ASTSTART
$ ICL
ICL> HELP ASTERIX
ICL> ASTERIX
ICL> ASTHELP
ICL> (any ASTERIX command)
```

from DCL:

```
$ ASTSTART
$ ASTHELP
$ (most, but not all, ASTERIX commands)
```

Demonstration:

A demonstration session is described in SUN/98.

4.2 CCDPACK — CCD data reduction [SUN/139]

A package to perform the initial processing of CCD data. Its main advantage over previous methods is its enhanced functionality. It processes large amounts of data easily and efficiently, with a minimum of effort on the user's part.

It includes routines for performing:

- Bias calibration data preparation.
- Bias subtraction.
- Flash and dark calibration data preparation.
- Flash and dark-count correction.
- Flatfield data preparation.
- Flatfield correction.

The following features are of particular note:

- Accesses lists of NDFs, using wildcards and by using names stored in text files.
- Logs the progress of a reduction sequence.
- Processes all non-complex numeric HDS data types.
- Supports many data-combination techniques.
- Takes full account of statistical uncertainty using variance production and propagation.

To run CCDPACK:

from ICL:

```
ICL> HELP CCDPACK
ICL> CCDPACK
ICL> <individual commands>
ICL> HELP <command>
```

from DCL:

```
$ CCDPACK
$ CCDHELP
$ (any CCDPACK commands)
```

4.3 CONVERT — Data format conversion [SUN/55]

This package converts data between the Starlink standard *n*-dimensional data format (NDF) and other formats. Currently, it can handle three data formats:

- DIPSO.
- FIGARO (version 2).
- INTERIM (BDF).

To run CONVERT:

from ICL:

```
ICL> CONVERT
ICL> (any of the CONVERT programs)
```

from DCL:

```
Simply type the name of the conversion program you want.
```

Demonstration:

This will convert an NDF file into BDF format, as used by the earlier Interim environment.

```
ICL> CONVERT
ICL> NDF2BDF
NDF - Name of NDF to be converted > adam_examples:image
BDF - BDF filename > image
ICL>
```

A file IMAGE.BDF will have been created in your current directory.

4.4 DAOPHOT — Stellar photometry [SUN/42]

A *stellar photometry* package written by Peter Stetson at the Dominion Astrophysical Observatory, Victoria, B.C., Canada and adapted for use under ADAM. It performs the following tasks:

- Finding objects.
- Aperture photometry.
- Obtaining the point spread function.
- Profile-fitting photometry.

Profile fitting in crowded regions is performed iteratively, which improves the accuracy of the photometry. It does not directly use an image display (which aids portability), although three additional routines allow results to be displayed on an image device. It uses image data in NDF format.

To run DAOPHOT:

from ICL:

```
Cannot be run from ICL.
```

from DCL:

```
$ DAOPHOT
Command: HELP
Command: ...
Command: EXIT
```

N.B. If you enter 'DAOPHOT' just to see what happens, you will be disappointed to receive the message `'Value unacceptable -- please re-enter'`. You will need to read the documentation and user manual before you can make any progress with this program.

4.5 FIGARO — General spectral reduction [SUN/86]

This is a *general data reduction* system written by Keith Shortridge at Caltech and the AAO. Most people find it of greatest use in the reduction of spectroscopic data, though it also has powerful image and data cube manipulation facilities. Starlink recommends FIGARO as the most complete spectroscopic data reduction system in the Collection. Examples of its facilities are:

- Analyse absorption lines interactively.
- Aperture photometry.
- Calibrate B stars.
- Calibrate flat fields.
- Calibrate using flux calibration standards.
- Calibrate wavelengths of spectra.
- Correct S-distortion.
- Extract spectra from images and images from data cubes, and insert spectra into images and images into data cubes.
- Extract spectra from images taken using optical fibres.
- Fit Gaussians to lines in a spectrum interactively.
- Generate and apply a spectrum of extinction coefficients.
- Input, output, and display data.
- Look at the contents of data arrays, other than graphically.
- Manipulate complex data structures (mainly connected with Fourier transforms).
- Manipulate data arrays 'by hand'.
- Manipulate images and spectra (arithmetic and more complicated).
- Process data taken using FIGS (the AAO's Fabry-Perot Infra-Red Grating Spectrometer).
- Process echelle data, in particular the UCL echelle in use at the AAO.

At present, a number of related packages are bundled with FIGARO. In future, these may be released as separate items. They include:

TWODSPEC [SUN/16]

This reduces and analyses long-slit and optical-fibre array spectra. A number of its functions are useful outside the area of spectroscopy. The main application areas are:

- Line profile analysis; LONGSLIT analyses calibrated long-slit spectra. For example, it can fit Gaussians, either manually or automatically, in batch. It can handle data with two spatial dimensions, such as TAURUS data. FIBDISP provides further options useful for such data, although it is primarily designed for fibre array data. An extensive range of options is available, especially for output.
- Two-dimensional arc calibration.
- Geometrical distortion correction; S-distortion and Line curvature.
- Conversion between FIGARO and IRAF data formats.
- Display programs.
- Removing continua.

To run FIGARO:

from ICL:

```
ICL> HELP FIGARO
ICL> FIGARO
ICL> HELP FIGARO CLASSIFIED
ICL> (any FIGARO commands)
```

from DCL:

```
$ FIGARO
$ HELP FIGARO
$ HELP FIGARO CLASSIFIED
$ (any FIGARO commands)
```

4.6 IRCAM — Infrared camera data reduction [SUN/41]

This package reduces, displays, and analyses 2-dimensional images from the *UKIRT infrared camera* (IRCAM). The image data reduction facilities available are:

- Mathematical and statistical operations.
- Size changing and mosaicking.
- Inspection.
- Interpolation.
- Smoothing.
- Feature enhancement.
- Bad pixel removal.
- Polarimetry.
- Median filtering of flat-fields.

The graphics and image display facilities available are:

- Image display of various types (PLOT, CONTOUR, NSIGMA, RANPLOT).
- Display cursor position and value.
- Colour control.
- Line graphics such as 1-dimensional cuts/slices through images and contour maps.
- Annotation.

To run IRCAM:

from ICL:

```
Cannot be run from ICL (it uses the older ADAMCL).
```

from DCL:

```
$ IRCAM_SETUP
$ IRCAM_CLRED
  (you are then asked to say where your data are
   and which plotting device you want to use)
Ircam-CLRED : > ?
  (produces a list of commands)
Ircam-CLRED : > (select commands)
    .
Ircam-CLRED : > EXIT
```

4.7 KAPPA — Kernel applications [SUN/95]

The Kernel Application Package runs under ADAM, using the NDF data format, and provides *general-purpose applications*. It is the backbone of the software reorganization around the ADAM environment, and its applications integrate with other packages such as PHOTOM, PISA, and FIGARO. It is usable as a single large program from the ADAM command language ICL, or as individual applications from DCL.

It handles bad pixels, and processes quality and variance information within NDF data files. Although oriented towards image processing, many applications will work on NDFs of arbitrary dimension. Its graphics are device independent. Currently, KAPPA has about 140 commands and provides the following facilities for data processing:

- Generation of NDFs and ASCII tables by up-to-date FITS readers.
- Generation of test data, and NDF creation from ASCII files.
- Setting NDF components.
- Arithmetic, including a powerful application that handles expressions.
- Editing pixels and regions, including polygons and circles, and re-flagging bad pixels by value or by median filtering.
- Configuration changing: flip, rotate, shift, subset, dimensionality.
- Image mosaicking; normalization of NDF pairs.
- Compression and expansion of images.
- Filtering: box, Gaussian, and median smoothing; very efficient Fourier transform, maximum-entropy deconvolution.
- Surface fitting.
- Statistics, including ordered statistics, histogram, pixel-by-pixel statistics over a sequence of images.
- Inspection of image values.
- Centroiding of features, particularly stars; stellar PSF fitting.
- Detail enhancement via histogram equalization, Laplacian convolution, edge enhancement via a shadow effect, thresholding.

There are also many applications for data visualization:

- Use of the graphics database, AGI, to pass information about pictures between applications. Facilities for the creation, labelling and selection of pictures, and obtaining world and data co-ordinate information from them.
- Image and greyscale plots with a selection of scaling modes and many options such as axes.
- Creation, selection, saving and manipulation of colour tables and palettes (for axes, annotation, coloured markers and borders).
- Snapshot of an image display to hardcopy.
- Blinking and visibility of image-display planes.
- Line graphics: contouring, including overlay; columnar and hidden-line plots of images; histogram; line plots of 1-d arrays, and multiple-line plots of images; slices through an image. There is some control of the appearance of plots.

To run KAPPA:

See Chapter 3 for a demonstration of KAPPA.

4.8 PHOTOM — Aperture photometry [SUN/45]

This performs aperture photometry. It has two basic modes of operation:

- Using an interactive display to specify the positions for the measurements.
- Obtaining those positions from a file.

The aperture is circular or elliptical, and the size and shape can be varied interactively on the display, or by entering values from the keyboard or parameter system. The background sky level can be sampled interactively by manually positioning the aperture, or automatically from an annulus surrounding the object.

To run PHOTOM:

from ICL:

```
ICL> HELP PHOTOM
ICL> PHOTOM
IN - NDF containing input image /@ramp1/ > adam_examples:image
COMMAND - PHOTOM /'Values'/ > H (for help)
COMMAND - PHOTOM /'Values'/ > (an option from the menu)
COMMAND - PHOTOM /'Values'/ > E
```

from DCL:

```
$ RUN PHOTOM_DIR:PHOTOM
IN - NDF containing input image /@ramp1/ > adam_examples:image
.
. (as above)
```

4.9 PISA — Object finding and analysis [SUN/109]

The Position, Intensity and Shape Analysis package, PISA, locates and parameterizes objects in an image frame. The core of the package is a routine which performs image analysis on a 2-dimensional data frame. It searches for objects having a minimum number of connected pixels above a given threshold, and extracts the image parameters (position, intensity, shape) for each object. The parameters can be determined using thresholding techniques, or an analytical stellar profile can be used to fit the objects. In crowded regions, deblending of overlapping sources can be performed.

The package derives from the APM IMAGES routine originally written by Mike Irwin at the University of Cambridge to analyse output from the Automatic Photographic Measuring system.

To run PISA:

from ICL:

```
ICL> HELP PISA
ICL> PISA
ICL> <individual commands>
ICL> HELP <command>
```

from DCL:

```
$ PISA
$ <individual commands>
$ HELP <command>
```

Demonstration:

This example performs isophotal analysis with deblending of overlapped images on a frame containing a mixture of stars and galaxies. The results are then plotted on a suitable device.

```
ICL> PISA
     Welcome to PISA ...
ICL> PISAFIND
IN - NDF containing input image /.../ > PISA_DIR:FRAME
 Analysing whole image
MINPIX - Minimum pixel size for images (typically 4-16) > 6
METHOD - Intensity analysis ( 0=Isophotal, 1=Total, 2=Profile ) /0/ > 0
 Estimated background level =   492.2
 Background standard deviation =     7.4
BACKGROUND - Background (global sky) value /492.17/ >
THRESH - Threshold for analysis (data units) /18.61135/ >

 Total number of positive images = 118
 The results have been written to PISAFIND.DAT
     and PISASIZE.DAT
ICL> PISAPLOT
RESULTS - File of PISAFIND parameterised data /@PISAFIND.DAT/ >
DEVICE - Name of graphics device /@IKON/ >
ICL>
```

4.10 SCAR — Star catalogue database system [SUN/70, 106]

The Starlink Catalogue Access and Reporting system is a relational database management system. It was designed principally for extracting information from astronomical catalogues, but it can be used to process any data stored in relational form. A large number of catalogues are available, including the IRAS catalogues. For general database requirements, REXEC may be preferable. SCAR can perform the following functions:

- Extract data from a catalogue using selection criteria.
- Manipulate data using various statistical and plotting routines.
- Output data from a catalogue.
- Put a new catalogue into the database.
- Search catalogues and generate reports on what has been found.
- Sort, merge, join, and difference catalogues.
- Plot sources in a gnomonic (tangent plane) or Aitoff (equal area) projection.
- Analyse the fields of a catalogue by scatterplot and histogram.
- Calculate new fields.

A distinctive feature of SCAR is the use of index files which you can create and which contain pointers to rows in one or more catalogues. This is a compact and flexible method of accessing catalogues; for example, a very large catalogue may be physically ordered by declination, but you can create an index giving access to it ordered by flux.

To run SCAR:

from ICL:

```
$ SCARSTART
.
ICL> HELP SCAR
ICL> SCAR
ICL> SCAR_HELP  (for help on SCAR)
ICL> CAR_HELP   (for help on CAR commands)
ICL> CAT_HELP   (for help on catalogues)
ICL> (any SCAR commands)
```

from DCL:

```
$ SCARSTART
$ (any SCAR commands)
```

Demonstration:

A script is available which demonstrates some of the features of SCAR. It can be invoked by typing:

```
ICL> LOAD SCAR_DOC_DIR:SCAR_SCRIPT
```

It is suggested that you run this in an empty directory so that you can identify the files which have been created.

4.11 SPECDRE — Spectroscopy data reduction [SUN/140]

A package for spectroscopy data reduction and analysis. It fills the gap between FIGARO and KAPPA — on the one hand, all its routines conform with Starlink's concept of bad values and variances, on the other hand, they offer spectroscopy applications hitherto available only in FIGARO. In general, it can work on data sets with seven or less axes. Often an application will take just a one-dimensional subset as a spectrum. In most cases the spectroscopy axis can be any axis, but for some applications must be the first axis.

To run SPECDRE:

from ICL:

```
ICL> HELP SPECDRE
ICL> SPECDRE
ICL> <individual commands>
ICL> HELP <command>
```

from DCL:

```
<not specified>
```

4.12 SST — Simple software tools [SUN/110]

The Simple Software Tools package helps produce software and documentation, with particular emphasis on ADAM programming using Fortran 77. It performs fairly simple manipulations of software, and also tackles some of the commonly encountered problems which are not catered for in the more sophisticated commercial software tools (such as FORCHECK and VAXset) available on Starlink.

The main purpose of the first version is to extract information from subroutine 'prologues', and to format it to produce various forms of user documentation. A simple source-code and comment statistics tool is also included.

There are five applications:

- Convert 'old-Style' ADAM/SSE prologues to 'new-style' ones.
- Produce LaTeX documentation.
- Produce Help libraries.
- Produce STARLSE package definitions.
- Produce source-code statistics.

To run SST:

from ICL:

```
ICL> HELP SST
ICL> SST
ICL> HELP <any SST command>
ICL> (any of the SST commands)
```

from DCL:

```
$ SST
$ (any of the SST commands)
```

Demonstration:

This simple demonstration will create a file containing statistics about source and comment lines in a Fortran program; the one used happens to be the one which is run in the demonstration.

```
ICL> SST
ICL> FORSTATS
IN - Input file(s) /'*.FOR'/ > CONVERT_DIR:CONVERT.FOR
.
. (messages from FORSTATS)
.
ICL> $ TYPE FORSTATS.LIS
.
. (the output which was written by FORSTATS)
.
ICL>
```

4.13 TSP — Time-series and polarimetry analysis [SUN/66]

This is a data reduction package for time-series and polarimetric data. These facilities are missing from most existing data reduction packages which are usually oriented towards either spectroscopy or image processing or both. Currently TSP can process the following data:

- Spectropolarimetry obtained with the AAO Pockels cell spectropolarimeter in conjunction with either IPCS or CCD detectors.
- Time series polarimetry obtained with the Hatfield Polarimeter at either UKIRT or AAT.
- Time series polarimetry obtained with the University of Turku UBVRI polarimeter.
- Five channel time series photometry obtained with the Hatfield polarimeter at the AAT in its high speed photometry mode.
- Time series infrared photometry obtained with the AAO Infrared Photometer Spectrometer (IRPS).
- Time series optical photometry obtained using the HSP3 high speed photometry package at the AAT.

To run TSP:

from ICL:

```
ICL> HELP TSP
ICL> TSP
ICL> (any TSP command)
```

from DCL:

```
Cannot be run from DCL.
```

Demonstration:

The following example shows the use of the PPLOT command to plot a polarization spectrum. The SN1987A data file is included with the software, so you can use this command to check that TSP is working.

```
ICL> PPLOT
Loading TSP_DIR:TSP into xxxxTSP
INPUT - Stokes Data to Plot > TSP_DIR:SN1987A
BINERR - Error per bin (per cent) /0.1/ >
AUTO - Autoscale Plot /YES/ >
LABEL - Label for plot /''/ > SN1987A  1987 Sep 2
DEVICE - Plot Device > IKON
ICL>
```

Chapter 5
ICL — Interactive Command Language

5.1 Introduction

ICL is the most commonly used ADAM command language. It is designed to provide a programmable user interface to an astronomical data-reduction or data-acquisition system. In some ways, ICL is similar to a high level programming language such as Fortran or Pascal, but it has some important differences:

- It is an *interactive* language — It provides a complete environment for entering, editing and debugging programs in much the same way that Basic does, rather than relying on external editors, compilers and so on.
- It is a *command* language — One of its main uses is to enable you to type commands with few format restrictions. For example, it can be used to run the FIGARO data reduction system and it is possible to type FIGARO commands in almost exactly the same format as was previously used from DCL.
- It is a *programming* language — However it is intended only for writing relatively simple and straightforward programs. Its requirements differ from those of most modern programming languages which are designed for the needs of big software projects. ICL is designed to make simple programs easy to write.

The definitive reference manual and user's guide is SG/5. The language is summarised in Chapter 19. ICL also has an on-line Help system — simply type HELP.

5.2 Starting and stopping ICL

ICL is started by typing:

```
$ ICL
```

(Always remember that you must previously have executed SSC:LOGIN and ADAMSTART.)

ICL is stopped by issuing the EXIT command:

```
ICL> EXIT
```

If things are hopeless you can, of course, abandon ship by typing Ctrl/Y.

5.3 Modes

ICL can be used in two different *modes*:

- Direct Mode
- Procedure Mode

In *Direct Mode*, statements are executed immediately after they are typed in. In *Procedure Mode*, statements are first entered into a procedure, and subsequently executed by running that procedure. Another difference between the modes is that Control Statements (see below) can only be executed in Procedure Mode.

5.4 Statements

The *statements* of the language can be grouped into two sets:

- Direct Statements
- Control Statements

Direct statements are executed immediately and can be entered in either mode. *Control statements* affect the order in which statements are executed and can only be executed in Procedure mode. You can also enter *Comments* which are lines whose first non-blank character is '{'. Comments are ignored by the language interpreter. If desired, they can be terminated by '}', but this isn't necessary and is a matter of taste. Here is an example of a comment:

```
{ This is a comment
```

Although comments can be entered in Direct mode, their real purpose is to document procedures written in Procedure mode.

When entering statements in ICL, all the normal command line editing facilities available in DCL may be used. The Up arrow or ctrl/B keys may be used to recall previous commands, and the Down arrow key will step to the next command. Any command back to the start of the ICL session may be recalled.

A command may take more than one line. A tilde (~) symbol at the end of a line is used to indicate that the command continues on the next line.

5.5 Direct statements

There are three types of direct statements:

Immediate Statement:

This is used to make ICL do simple calculations. It consists of an equals sign (=) followed by an expression, and causes the value of the expression to be printed on the terminal. For example:

```
ICL> =1+2+3
         6
ICL> =SQRT(2)
1.414214
ICL>
```

Assignment Statement:

This is exactly the same as the Fortran assignment statement and is used to assign a value to an ICL variable. For example:

```
ICL> PI=3.1415926
ICL> =2*pi
6.283185
ICL>
```

Note that the case of letters doesn't matter. 'PI' and 'pi' are the same variable. In practice, names like 'PI' are the names of HDS objects which store the constant values.

Command:

A command consists of a command name followed (optionally) by one or more parameter specifications:

command_name [parameter_specification] . . .

The parameter specifications may be separated by commas, or by one or more spaces. The forms that parameters can take are described in Chapter 8. Here are some examples of commands:

```
PRINT Single Root is (-C/B)
QUADRATIC 7.2,18,4.6
TESTR 5.8
```

Commands are a very important part of ICL and several types exist. You can't tell which type a command is just by looking at it; you have to know how it is defined. Commands are described in more detail in Chapter 6.

All three types of direct statement make use of *expressions*. These are described next.

5.6 Expressions

One of the basic syntactical units of the ICL language is the *Expression*. Expressions are built up by operating on *values* using *operators*:

VALUE *operator* VALUE *operator* VALUE . . .

Here are some examples of expressions:

```
2
X
X+5
Y*SIN(THETA)
```

The value of an expression belongs to one of the following four *types*:

- Real — same as Fortran DOUBLE PRECISION type.
- Integer — same as Fortran INTEGER type.
- Logical — same as Fortran LOGICAL type.
- String — strings of characters, similar to Fortran CHARACTER*n type.

The two constituents of expressions — *value* and *operator* — will now be considered in more detail:

Value — A value can be represented in three ways:

Constant:

Real and *Integer* constants are represented by numbers written in the same formats that are accepted in Fortran. *Integer* constants may also be entered in binary, octal, or hexadecimal format by preceeding the value with `%B`, `%O`, or `%X` respectively.

Logical constants are written as `TRUE` or `FALSE`. Note that they are not delimited by decimal points as in Fortran.

String constants consist of any sequence of characters enclosed in either single or double quotes. Two consecutive quote symbols in a string are used to represent a single quote. String constants are the one place in ICL where the case of letters is significant.

Here are some examples of constants:

```
Real:          1.234E-5              3.14159
Integer:       123      %B100110     %O377      %Xffff
Logical:       TRUE     FALSE
String:        'This is a string'    "So is this"      ''
```

The last example defines a string of zero length (valid in ICL, but not in Fortran).

Variable:

Variables are represented by names composed of characters which may be letters, digits, or the underscore character (_). The first character must be a letter. The first 15 characters of a variable name are significant.

An important difference between ICL and Fortran is in the handling of variable *types*. In Fortran, each variable name has a unique type associated with it, and this type is either derived implicitly from the first letter of the name, or is explicitly specified in a declaration. In ICL, variable names do not have types — only values have types. A variable gains a type when it is assigned a value. This type can change when a new value is assigned to it. Thus we can have the following series of assignments making the variable X an integer, real, logical, and string in sequence:

```
ICL> X = 123
ICL> X = 123.456
ICL> X = TRUE
ICL> X = 'String'
```

This approach to variable types means that we do not have to declare the variable names we use, which helps to keep programs simple[1].

Function:

ICL provides a variety of standard functions. They are written as in Fortran, and all the standard Fortran 77 generic functions which are relevant to the ICL data types are provided with the same names. Thus SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2, LOG, LOG10, EXP, SQRT and ABS are all valid functions. A complete list of functions is given in Chapter 19.

[1]The disadvantage is that ICL cannot usually spot cases where we accidentally mistype a variable name, as can languages which enforce declaration of variables (such as Fortran with the IMPLICIT NONE directive). This is not thought to be a serious problem for the relatively simple programs for which ICL is intended.

Operator —

The operators which are used to build up expressions are listed in the following table in order of priority:

```
1 (highest)            **
2                      *   /
3                      +   -
4                      =   >   <   >=  <=  <>  :
5 (lowest)             NOT   AND   OR   &
```

The order of evaluation of expressions is determined by the priority of the operators. The rules are the same as those in Fortran[2], with arithmetic operators having the highest priority and logical operators the lowest. This means that a condition such as $0 < X + Y \leq 1$ can be expressed in ICL as follows:

```
X+Y > 0  AND  X+Y <= 1
```

without requiring any parentheses. However, in general it is good practice to use parentheses to clarify any situation in which the order of evaluation is unclear.

The '&' operator performs string concatenation. If one of its operands is a numeric value, it will be converted to a string. The ':' operator is used for formatting numbers into character strings as described in Section 6.4

In evaluating expressions, ICL will freely apply type conversion to its operands in order to make sense of them. This means not only that integers will be converted to reals when required, but also that strings will be converted to numbers when possible. A string can be converted to a number if the value of the string is itself a valid ICL expression; for example, the string `'1.2345'` has a numeric value. The string `'X+1'` has a numeric value if X is currently a numeric variable (or if X is a string which has a numeric value).

[2]But different from those in Pascal.

Chapter 6
ICL Commands

6.1 Commands

In the last chapter we introduced the *Command* as one of the Direct Statements of the ICL language. You will remember that the format was:

> command_name [parameter_specification] ...

There are two main types of commands:

- Built-in commands
- User-defined commands

The difference between them is that built-in commands are always available as they are an intrinsic part of the language, while user-defined commands are only available if you have defined them yourself in some way. The two types will be considered separately in the following two sections.

If you define commands, or use an application that defines commands, there is a danger of ambiguity since you may give a new meaning to an existing command name. Just for the record, here is the search-path that ICL uses when it searches for command definitions:

- User-defined: DEFSTRING, DEFUSER, DEFPROC
- Procedures: PROC
- Built-in commands to control ADAM tasks: ALOAD, ...
- Applications: DEFINE
- Other Built-in

Don't worry about those command names, you will learn about them latter.

6.2 Built-in commands

The built-in commands are listed in Chapter 19. Here is a classification:

Information & Escape — Chapter 5
HELP, EXIT

Defining user commands — Chapter 6
DEFSTRING, DEFINE, DEFUSER, DEFPROC

I/O — Chapter 6

Terminal
PRINT, INPUTt

Screen mode
SET SCREEN, SET NOSCREEN, SET ATTRIBUTES, LOCATE, CLEAR

Keyboard facilities
KEY, KEYTRAP, KEYOFF

Files
CREATE, OPEN, APPEND, CLOSE, WRITE, READt

Access to DCL — Chapter 6
$, SPAWN, DEFAULT,
ALLOC, DEALLOC, MOUNT, DISMOUNT

Parameters — Chapter 8
SETPAR, GETPAR,
CREATEGLOBAL, SETGLOBAL, GETGLOBAL

Procedures — Chapter 9
LIST, PROCS, VARS, EDIT, SET EDITOR,
SAVE, LOAD, DELETE,
SET TRACE, SET NOTRACE

Errors & Exceptions — Chapter 9
SIGNAL

HELP system — Chapter 9
DEFHELP

All these commands are described in the chapter specified.

6.3 User-defined commands

DEFSTRING	Associate command with equivalence string
DEFINE	Define command to run a program
DEFUSER	Define command to run a subroutine
DEFPROC	Define command to run an ICL procedure

There are three types of command which you can use to define your own commands:

- To define commands which stand for strings — DEFSTRING.
- To define commands which run programs — DEFINE, DEFUSER.
- To define commands which run ICL procedures — DEFPROC.

These are described briefly below:

DEFSTRING — associates a command with an equivalence string:

```
DEFSTRING  command  equivalence-string
```

when you enter `command`, it is as if you had entered `equivalence-string`.

DEFINE — defines a command which causes a program to be executed:

```
DEFINE  command  program-name
```

When you enter `command`, program `program-name` is executed.

DEFUSER — defines a command which causes a compiled subroutine to be executed:

```
DEFUSER  command  image-name
```

When you enter `command`, the shareable image `image-name` is executed.

DEFPROC — defines a command which causes an ICL procedure to be executed:

```
DEFPROC  command  file
```

When you enter `command`, the ICL procedure stored in file.ICL is compiled (first time only) and executed.

The DEFINE command is explained further in Chapter 7, and the definition and use of ICL procedures is explained in Chapter 9.

6.4 Input/Output commands

The I/O commands fall into three classes:

- Terminal
- File
- Screen mode

These are described in the following sub-sections.

6.4.1 Terminal

INPUT	Input string from terminal
INPUTI	Input integers from terminal
INPUTL	Input logicals from terminal
INPUTR	Input reals from terminal
PRINT	Output to terminal

Input:

The commands for terminal input are INPUT, INPUTR, INPUTI, and INPUTL. Command INPUT reads a line of text from the terminal into a string variable. An example is:

```
ICL> INPUT Enter your name> (X)
Enter your name>Mike
ICL> PRINT (X)
Mike
ICL>
```

The last parameter must specify a variable in which the input will be stored and must, therefore, be in parentheses. The earlier parameters form a prompt string.

Commands INPUTR, INPUTI, and INPUTL are used to input real, integer, and logical values. A single command may be used to supply values for more than one variable, so these commands have the general form:

```
ICL> INPUTR  Prompt  (X)  (Y)  (Z)  ...
```

For these three commands, only the first parameter is used to provide the prompt string, so if it has spaces in it, the string must be enclosed in quotes. For example:

```
ICL> INPUTR 'Give values of X and Y > ' (X) (Y)
Give values of X and Y > 2.3 8.9
ICL> PRINT (X) (Y)
2.300000 8.900000
ICL>
```

INPUTL will accept values of TRUE, FALSE, YES, and NO in either upper or lower case, as well as abbreviations.

Output:

You can output to the terminal by using the PRINT command:

```
PRINT  p1  p2  ...
```

The parameters are concatenated and printed on the terminal. For example

```
ICL> X=2
ICL> PRINT The square root of (X) is (SQRT(X))
The square root of          2 is 1.414214
```

Another way of writing to the terminal is by using an Immediate statement:

```
ICL> =1+2+3
         6
```

Formatting Output:

While Fortran regards formatting of numbers for output as part of an output operation, ICL performs formatting using an operator (:) which produces a string result from a numeric operand. Thus, if I is an integer variable, the expression `I:5` has as its value the string which is produced by converting I with a field width of 5 characters. It is equivalent to an I5 format in Fortran. Similarly, if X is a real variable, the expression `X:10:4` produces the value of X formatted in a Fortran F10.4 format (i.e. a field width of 10 characters with 4 decimal places). The ICL formatting is not precisely equivalent to the Fortran form because ICL will extend the field width if a number is too large to fit in the requested width. For example:

```
ICL> =1.234567:5:2
 1.23
ICL> =12.34567:5:2
12.35
ICL> =123.4567:5:2
123.46
ICL> =123456.7:5:2
123456.70
ICL>
```

Integers can also be formatted in binary, octal, decimal, or hexadecimal formats using the functions BIN, OCT, DEC, and HEX. These have the form `HEX(X,n,m)` which would return a string of n characters containing the number X with m significant digits. Parameters n and m may be omitted, in which case they default to the number of digits needed to represent a full 32 bit word. Using these forms together with constants in various bases, ICL can be used to perform conversions between bases. For example:

```
ICL> =%Xffff
      65535
ICL> =hex(65535)
 0000FFFF
ICL> =oct(%XFF,5,5)
00377
ICL>
```

6.4.2 Screen mode

SET SCREEN	Select screen mode
SET NOSCREEN	Select normal mode
SET ATTRIBUTES	Set attributes for text written with LOCATE
LOCATE	Write to screen at specified position
CLEAR	Clear range of lines

Screen mode allows more control over the terminal screen than is possible in the normal mode. It also allows more control over the use of the keyboard. Screen mode is implemented using the DEC screen management (SMG$) routines of the run time library, and will work on any terminal compatible with these routines.

Screen mode is selected by the SET SCREEN command. In this mode, the terminal screen is divided into an upper fixed region and a lower scrolling region. The size of the scrolling region may be specified by an optional parameter to SET SCREEN.

```
ICL> SET SCREEN 10
```

will select 10 lines of scrolling region. The size of the scrolling region can be changed by further SET SCREEN commands. SET NOSCREEN is used to leave screen mode and return to normal mode.

Standard terminal I/O operations work exactly as normal in the scrolling region of the screen. However, an additional facility is the ability to examine text which has scrolled off the top of the scrolling region. The Next Screen and Prev Screen keys on a VT200 (or ctrl/N, ctrl/P on other terminals) may be used to move through the text. Any output since screen mode was started is viewable in this way.

To write to the fixed part of the screen, the command LOCATE is used:

```
ICL> LOCATE 6 10   This text will be written starting at Row 6 Column 10
```

The first two parameters specify the row and column at which the text will start. The remaining parameters form the text to be written.

The SET ATTRIBUTES command provides further control over text written with the LOCATE command. This command has a parameter string composed of any combination of the letters R (Reverse Video), B (Bold), U (Underlined), F (Flashing) and D (Double Size). These attributes apply to all LOCATE commands until the next SET ATTRIBUTES command. SET ATTRIBUTES with no parameter gives normal text.

The CLEAR command is used to clear all or part of the fixed region of the screen. For example:

```
ICL> CLEAR 6 10
```

clears lines 6 to 10 of the screen.

6.4.3 Keyboard facilities

KEY	Define equivalence string for key
KEYTRAP	Specify trapping of key in a procedure
KEYOFF	Turn off trapping of key in a procedure

The KEY command may be used to define an equivalence string for any key on the keyboard. For example:

```
ICL> KEY  PF1  PROCS#
```

defines the PF1 key so that it issues the PROCS command. The # character is used to indicate a Return character in the equivalence string.

The name of a main keyboard key may be specified as a single character or as an integer representing the ASCII code for the key. Keypad and function keys are specified by names as follows:

```
Keypad keys             PF1, PF2, PF3, PF4, KP0, KP1, KP2, KP3, KP4, KP5,
                        KP6, KP7, KP8, KP9, ENTER, MINUS, COMMA, PERIOD.

Function Keys (VT200)   F6, F7, F8, F9, F10, F11, F12, F13, F14, HELP,
                        DO, F17, F18, F19, F20.

Editing Keypad (VT200)  FIND, INSERT_HERE, REMOVE, SELECT, PREV_SCREEN, NEXT_SCREEN.

Cursor Keys             UP, DOWN, LEFT, RIGHT.
```

The KEYTRAP command and the INKEY() function allow an ICL procedure to test for keyboard input during its execution, without having to issue an INPUT command and thus wait for input to complete.

KEYTRAP specifies the name of a key to be trapped.

KEYOFF turns off the trapping of a specified key.

INKEY() is an integer function which returns zero if no key has been pressed, or the key value if a key has been pressed since the last call. The key value is the ASCII value for ASCII characters, or a number between 256 and 511 for keypad and function keys.

KEYVAL(S) is a function which obtains the value from the key name.

Here is an example of the use of these commands and functions within a procedure:

```
{ Trap ENTER and LEFT and RIGHT arrow keys

KEYTRAP ENTER
KEYTRAP LEFT
KEYTRAP RIGHT

LOOP
  K = INKEY()
  IF K = KEYVAL('ENTER') THEN
```

```
    .
    ELSE IF K = KEYVAL('LEFT') THEN
    .
    ELSE IF K = KEYVAL('RIGHT') THEN
    .
    ELSE
    .
    ENDIF
  END LOOP
```

6.4.4 File

CREATE	Create new file and open for output
OPEN	Open existing file for input
APPEND	Open existing file for output, append text
CLOSE	Close file
READ	Read line from file
READI	Read integers from file
READL	Read logicals from file
READR	Read reals from file
WRITE	Write to file

ICL can read and write text files. In order to access such files, they must first be opened using one of the commands CREATE, OPEN, or APPEND, for example:

```
  ICL> CREATE MYFILE
```

will create a file called `MYFILE` and open it for output — `MYFILE` is the name used within ICL for the file. In this case, the file will appear in your default directory as `MYFILE.DAT`. However, the file name may be specified explicitly by adding a second parameter to the commands, for example:

```
  ICL> OPEN  INFILE  DISK$DATA:[ABC]FOR008.DAT
```

opens an existing file for input and the file is known internally as `INFILE`.

The APPEND command opens an existing file for output. Anything written to it is appended to the existing contents.

A line of text is written to a file with the WRITE command. WRITE is similar to PRINT, the only difference being that its first parameter specifies the internal name of the file to which the data will be written.

Text is read from files with the commands READ, READR, READI, and READL. These are analogous to the INPUT commands for terminal input. The first parameter specifies the internal name of the file. READ reads a line of text into a single string variable. The other commands read one or more real, integer, or logical values.

When a file is no longer required it may be closed using the CLOSE command which has a single parameter: the internal name of the file.

The following example defines a procedure which uses these commands to read a file containing three real numbers in free format and output the same numbers as a formatted table.

```
PROC REFORMAT
{ Open input file and create output file
  OPEN INFILE
  CREATE OUTFILE
{ Copy lines from input to output
  LOOP
    READR INFILE (R1) (R2) (R3)
    WRITE OUTFILE (R1:10:2) (R2:10:2) (R3:10:2)
  END LOOP
END PROC
```

Note that no specific test for completion of the loop is included. When an end-of-file condition is detected on the input file, the procedure will exit and return to the `ICL>` prompt with an appropriate message[1].

6.5 DCL commands and VMS processes

6.5.1 Executing DCL commands

$	Execute a DCL command in $'s subprocess
SPAWN	Execute a DCL command in new subprocess

When using ICL, it is frequently useful to be able to access features of Digital's command language DCL. Typical operations we may want to do include listing directories, copying files, allocating tape drives, and mounting tapes. The ICL command '$' allows any DCL command to be issued from inside ICL. Its form is simply:

```
ICL> $ dcl_command
```

where `dcl_command` is any command we could issue from the DCL '$' prompt. For example:

```
ICL> $ COPY *.SDF DATADIR:*.SDF
ICL> $ RUN MYPROGRAM
```

There is one restriction — you must use a complete DCL command. You can't, for example, just type `$ COPY` and expect DCL to prompt you for the two file specifications. Apart from this, any command acceptable to DCL can be issued in this way.

There is a way around the restriction on prompts mentioned above. This is to use the command 'SPAWN' rather than the command 'DCL'. For example:

```
ICL> SPAWN COPY
_From: *.SDF
_To: DATADIR:*.SDF
```

in which case you get the '_From:' and '_To:' prompts, just as you do in normal DCL. The disadvantage of SPAWN is that it is slower. This is because SPAWN creates a new subprocess to run each command, whereas DCL creates a permanent subprocess in which all commands are run.

SPAWN has another use — by just typing SPAWN you can get a DCL '$' prompt from which a series of DCL commands can be executed. LOGOUT is used to return control to ICL.

[1]A tidier exit can be arranged by using an exception handler for the EOF exception, see Chapter 9.

6.5.2 Processes and subprocesses

In VMS, a program runs in a *process*. If that program wishes to run another program or do something else, it must first create a *subprocess*. ICL is such a program and it runs application programs. The applications thus run in subprocesses, and so do DCL commands issued from ICL. This is the heart of many of the sometimes unexpected properties of ICL.

VMS is not particularly efficient at starting up processes, so loading a program takes time. ADAM deals with this through a structure called a monolith (discussed in Chapter 7).

The worst problems with subprocesses arise when something goes wrong. Uninhibited use of such things as ctrl/C to kill an application can have serious consequences for ICL, resulting in a loss of context and the consequent waste of time getting back to where you started. Ctrl/C returns you to the ICL prompt, while ctrl/Y takes you right out of ICL and back into DCL. A better way to get out of a program is to enter the *abort* response '!!' when prompted for a parameter (see Chapter 8.)

Note that ctrl/C will only break out of a program and return you to ICL if the program is run as a subprocess of ICL. This is usually the case, but will not be if the command being executed was defined using DEFUSER (as in ASTERIX for example).

6.5.3 Changing your default directory

DEFAULT	Set default directory

You can change your default directory by using the DCL command 'SET DEFAULT'. This will change the default directory of the DCL subprocess, but not of the process running ICL. Thus, an additional ICL command DEFAULT has been provided. This changes the default directory of both the process running ICL and the DCL subprocess (if one exists). The format for specifying the directory is exactly the same as that accepted by the equivalent DCL command.

6.5.4 Managing tape drives

ALLOC	Allocate a device
DEALLOC	Deallocate a device
MOUNT	Mount a device
DISMOUNT	Dismount a device

Similar problems with processes occur when allocating and mounting tape drives. The DCL command 'ALLOCATE' will allocate the device to the DCL subprocess. This *may* be what you want; for example, if you are going to use another DCL command (such as 'BACKUP') to read or write the tape. However, if a tape is to be processed using a FIGARO command, say, it must be allocated to the process running ICL. A set of commands has been provided for this purpose.

ALLOC allocates a device and may specify a generic name; the name of the device actually allocated will be returned in the optional second parameter. For example:

```
ICL> ALLOC MT
_MTA0: Allocated
ICL> ALLOC MT (DEVICE)
_MTA1: Allocated
ICL> =DEVICE
_MTA1:
```

DEALLOC deallocates a device.

MOUNT performs a MOUNT/FOREIGN at the tape's initialised density. It does not provide the many qualifiers of the DCL command. There are several additional optional parameters for some of these commands.

DISMOUNT has an optional parameter which is used to specify that the tape be dismounted without unloading:

```
    ICL> DISMOU MTA1 NOUNLOAD
```

Chapter 7
Running Applications

7.1 ADAM applications

An ADAM application is a program which uses ADAM subroutines. From the user's point of view, the most important of these is the Parameter system since this controls (in conjunction with the user language) the interaction between the program and the user. The Parameter system uses a file called an *Interface File* which stores information about a program's parameters.

Many users will only use applications which have been written by someone else. In this case, everything necessary to enable them to be used should already have been carried out. This includes the preparation of the interface files and the definition of commands. All you need do is carry out a specified startup procedure. However, if you have written your own application, you will need to do some preparation of your own before using it.

7.2 The running process

The process of running an ADAM application can be summarized as follows

Read the documentation —
I know it's hard, but it often really does help to read about the program you intend to use. Program documentation often tells you what the program does, what data it uses, and what its parameters are and how to specify them. Why not try it?

ADAMSTART —
Don't forget to enter `ADAMSTART` — people do.

Select a command language —
The options are DCL, ICL, and SMS at the moment. ICL is the one we recommend. SMS is usually used only for data acquisition and is not covered in this guide. Some packages won't run under some languages — Chapter 4 tells you which ones.

Start the package —
Some packages need to have various definitions and logical names defined before you can use them. For example, before you can use ASTERIX you must enter the DCL symbol `ASTSTART`. Once again, Chapter 4 tells you what to do for specific applications.

Select programs —
Most applications contain many programs. You must select the programs you want to use. For applications provided by other people, this will normally involve just typing the name of the program. However, if you have written your own application, you will need to define the appropriate commands required to run your programs. Applications which have large numbers of programs can be executed more efficiently if they are grouped together into a *Monolith*. These are considered below.

Specify parameter values —
Once you start running applications you will need to control them. This is done by specifying parameter values — names of data files, values, options, and so on. This is the most complex part of the process and is discussed in Chapter 8.

7.3 Selecting programs

You normally select programs just by typing their names. However, for this to work, the program name must have been defined as a DCL symbol or an ICL command. Also, there might be a delay the first time you run a program in an application. This is usually because a large *Monolith* is being loaded.

7.3.1 Defining commands to run programs

To run programs under VMS, their names are defined as DCL symbols so that typing the name runs the program. The symbol definitions are executed by running a command procedure.

Similarly, to run programs under ICL, their names are defined as ICL commands using the DEFINE command described in Chapter 6. Usually, the command definitions are executed by running an ICL procedure which contains them. Before a program can be run under ICL, a command to run it must be defined.

7.3.2 Monoliths

Programs are sometimes stored together in structures called *monoliths* which are loaded for execution as a single entity. In this case a command must be defined for each program in the monolith. When you use an application package, the commands needed to execute its programs are normally defined for you when you start up the package. Thus:

```
ICL> KAPPA
```

will cause all the commands required to run the KAPPA programs to be defined. What happens is that a file called `KAPPA_DIR:KAPPA.PRC`[1] containing the required DEFINE commands is automatically input to ICL for execution. The commands look like this:

```
define add        kappa_dir:kappa
define aperadd    kappa_dir:kappa
        ...
define zaplin     kappa_dir:kappa
```

There are also commands to display the initial messages output by the package. `KAPPA.PRC` is an example of an ICL *Command File* which is described in more detail in Chapter 9. Notice that all the commands refer to the same executable image (the monolith), and that the directory in which this is to be found is specified explicitly so ADAM_EXE is not used.

Monoliths are useful because they enable lots of programs to be loaded within a single process creation. Since process creation is quite slow under VMS, you save time by doing it only once. The advantage of monoliths is that they reduce the time required to create processes. The disadvantage is the time required for the loading a large monolith instead of a small program; however, you only have to do it once.

[1] The file type '.PRC' is now obsolete. The normal file type for command files is '.ICL'.

Chapter 8
Specifying Parameter Values

8.1 Introduction

ADAM programs are controlled by specifying parameter values. You can specify them in two ways:

On a command line —
In this case, the value will be processed by a command language interpreter (DCL, ICL, or SMS) before being passed to the parameter system. So, for example, if you were talking to ICL you could include ICL functions in your values.

In response to a prompt —
In this case you are talking directly to the program through the ADAM parameter system, and you could not use ICL functions.

The thing to remember is that when you respond to prompts you are talking directly to the ADAM parameter system, whereas when you specify values on the command line, they will be processed by the language interpreter and then passed to the parameter system.

Usually, it is not necessary to specify parameters on the command line — you can just respond to prompts. However, there are three reasons why you may wish to do so:

- To avoid waiting for prompts.
- To set values for parameters which normally are not prompted.
- To use a command safely in a procedure.

8.2 Command line specification

This section refers to ICL commands. However, you can run ADAM programs by using DCL commands. These suffer from the following restrictions compared to ICL:

- It is not possible to use ICL expressions.
- It is not possible to return parameter values to the command language, and hence on to applications.
- Single quotes cannot be used, except where they are needed to force expansion of DCL symbols. Double quotes can be used.
- It is not possible to use an ! unless it is in a character string enclosed in double quotes. Normally this character indicates a comment.

There are two ways in which you can specify parameter values in an ICL command:

- By using a command which has been defined to run an ADAM program. (The definition is done by a DEFINE command.)
- By using the SETPAR command. (You can also store a parameter value in an ICL variable by using the analogous GETPAR command.)

The first way is the most usual and will now be considered further.

You will remember that a program's parameters may be assigned values on the command line:

```
ICL> command   p1   p2   ...
```

where `command` is a commmand that has been defined to run the program. This method is particularly useful for running programs in VMS batch mode, and for specifying values for parameters that would otherwise be defaulted. Normally, command-line specification will prevent prompting for a parameter's value unless there is an error in the given value, say giving a character string for a parameter of type _REAL.

There are two ways in which parameter values may be specified on the command line:

- by keyword
- by position

The two forms may not be mixed.

The keyword method is usually preferable because the position method has these disadvantages:

- Values must be given for *every* parameter preceding the last parameter for which a value is required; values for parameters after this one can be omitted.
- Some programs have many parameters and it is tedious to enter all the intermediate values between the ones you want to define, and difficult to remember them all.
- Some parameters may not have defined positions because they are normally defaulted.

8.2.1 Keyword method

In the keyword method, a parameter is identified by a *keyword* (specified in the program's interface file) and a value is assigned to it using the syntax:

```
keyword=value
```

The keyword is the name by which the parameter is known to the user. This is usually, but not necessarily, the same as the parameter name.

In some cases (such as when setting a switch) the keyword itself is all that needs to be specified: Here is an example:

```
ICL> DEFPIC CURPIC FRACTION=0.4
```

The keyword and data type corresponding to a particular parameter is specified in the program's interface file:

```
interface DEFPIC
  parameter FRACTION
    keyword 'FRACTION'
    type _REAL
    ...
  parameter CURPIC
    keyword 'CURPIC'
    type _LOGICAL
    ...
endinterface
```

Here, the keyword `CURPIC` is associated with a logical type parameter — just specifying its name sets its value to 'TRUE', specifying 'NOCURPIC' sets its value to 'FALSE'. The keyword `FRACTION` is associated with a real type parameter which is given the value 0.4. Keyword forms may appear in any order on the command line, and are ignored when calculating the 'position' of other parameters.

8.2.2 Position method

In the position method, the position of a value on the command line identifies the parameter which is being given that value. Here is an example:

```
 ICL> ADD HISIMAGE HERIMAGE THEIRIMAGE
```

The parameter corresponding to a particular position is specified in the program's interface file:

```
interface ADD
  parameter IN1
    position 1
    ...
  parameter IN2
    position 2
    ...
  parameter OUT
    position 3
    ...
  parameter TITLE
    position 4
    ...
endinterface
```

Here, the first parameter IN1 is given the value `HISIMAGE`, the second parameter IN2 is given the value `HERIMAGE`, and the third parameter OUT is given the value `THEIRIMAGE`. The fourth parameter TITLE is not given a value.

8.2.3 ICL variables and functions

Command-line specification can be used to put the values of ICL variables and functions into programs. A common situation where this is useful is when processing a sequence of files in the same way. For example, consider this ICL procedure:

```
PROC MULTISTAT
  LOOP FOR I=1 TO 10
    FILE = '@' & SNAME('REDX',I,2)
    STATS2D INPIC=(FILE) AGAIN=F XSTART=1 YSTART=1 XSIZE=62 YSIZE=58
  END LOOP
END PROC
```

This obtains the statistics of ten images named `REDX01`, `REDX02`, . . . , `REDX10`. The parameter with keyword INPIC is assigned the value of the ICL variable FILE. Variable FILE is in parentheses because syntactically it is an ICL expression. The ICL function SNAME is used to construct the appropriate file name, which must be preceded by an '`@`'.

Here is another example of the use of ICL variables in commands which uses KAPPA to 'flat field' a series of CCD frames:

```
PROC FLATFIELD
  INPUT Which flat field frame?: (FF)
  FF = '@' & (FF)
  INPUTI Number of frames:  (NUM)
  LOOP FOR COUNT=1 TO (NUM)
    INPUT Enter frame to flat field: (IMAGE)
    TITLE = 'Flat field of ' & (IMAGE)
    IMAGE = '@' & (IMAGE)
    IMAGEOUT = (IMAGE) & 'F'
    PRINT Writing to (IMAGEOUT)
    DIV IN1=(IMAGE) IN2=(FF) OUT=(IMAGEOUT) TITLE=(TITLE)
  END LOOP
END PROC
```

Here, the parameters of the DIV program are given values stored in ICL string variables.

8.3 Prompts and suggested values

If program parameters are not given values on the command line, values will be prompted for (unless this is suppressed). If the value supplied is not acceptable, the prompt will be repeated. If no acceptable value is supplied after five prompts, a 'Null value' is given to the program.

The general form of a prompt is:

```
<keyword> - <prompt> /<suggestion>/ >
```

However, the `/<suggestion>/` may not appear.

8.3.1 Prompts without a suggested value

The simplest type of prompt is one where the suggested value is omitted, for example:

```
LOW - Lower limit for data >
```

In this case, if you just press 'carriage return', the prompt will be repeated up to five times, and then a null value will be given to the program:

```
LOW - Lower limit for data >
LOW - Lower limit for data >
LOW - Lower limit for data >
...
```

You should specify the value you want, for example:

```
LOW - Lower limit for data > 1
```

This value will be accepted, and then the next prompt will be displayed.

8.3.2 Prompts with a suggested value

When a prompt includes a suggested value, you can override it by entering a value of your own. However, when this prompt appears again during a subsequent use of the program, the suggested value may or may not be the same as the value you last entered. Sometimes the current (last) value of the parameter is used, sometimes a fixed default value is used, and sometimes a dynamic default value is calculated by the program from the values of other parameters. It all depends on the characteristics of the program and its interface file. Parameter XDIM of program CREFRAME is an example of a parameter that uses the current value:

```
XDIM - x dimension of output array /64/ > 109
...
XDIM - x dimension of output array /109/ > 45
...
XDIM - x dimension of output array /45/ > 16
...
```

Current values of parameters are stored in a file in ADAM_USER, so they persist between ADAM sessions. The file should not be deleted unless the old values are not required.

Parameter TYPED of program CREFRAME is an example of a parameter that always offers the same suggested value, no matter what the current value is:

```
TYPED - Type of data to be generated /'GS'/ > RA
...
TYPED - Type of data to be generated /'GS'/ > BL
...
TYPED - Type of data to be generated /'GS'/ > GN
...
```

8.3.3 Global parameters

Normally, a program's parameter values are only accessible by that program. However, *global parameters* can be shared by many programs. The use of global parameters as suggested values can reduce the need for typing responses to prompts. For example, in this case:

```
ICL> creframe
...
OUTPIC - Image for output data > ramp4
ICL> look
INPIC - Image to be inspected/@ramp4/ >
...
```

the frame created by CREFRAME can be displayed by LOOK without having to retype its name. This is because parameter `OUTPIC` of program CREFRAME and parameter `INPIC` of program LOOK are both associated (in their interface files) with the global parameter `GLOBAL.DATA_ARRAY`. Thus, parameter `INPIC` offers as a suggested value the object name which was input as a value for the `OUTPIC` parameter of a different program.

In ICL you can create a global parameter using the CREATGLOBAL command, set its value using the SETGLOBAL command, and get its value using the GETGLOBAL command.

8.3.4 Missing prompts

Sometimes a prompt for a parameter value will not appear, even if you haven't specified a value on the command line. This is because the parameter system looks for a value for a parameter in a number of places, as specified in a 'search path' in the program's interface file. This feature is described in Chapter 14. A prompt will only appear if it cannot find a value in the search path, or if the search path specifically asks the parameter system to generate a prompt. This ability to supply values automatically enables programs with many options to avoid burdening the user unnecessarily with large numbers of prompts. If you want to supply your own value in these cases, you must specify a value on the command line, or demand to be prompted (see below).

When prompts don't appear, the program documentation may tell you what the unprompted parameters are, what options are available, and what the suggested values are. Another way is to look at the online help on a program's parameters; for example:

```
ICL> kaphelp creframe param *
```

gives details of all the parameters of the KAPPA program CREFRAME.

8.3.5 Keywords — RESET, PROMPT, ACCEPT

Another way in which prompts and suggested values can be controlled is by use of the keywords RESET, PROMPT, and ACCEPT in ICL commands.

RESET:

The RESET keyword causes the suggested value of all parameters (apart from those already specified before it on the command line) to be set to the original values specified by the program and interface file. For example, consider the prompt for the value of the XDIM parameter in the following successive executions of the CREFRAME program:

```
ICL> creframe
XDIM - x dimension of output array /64/ > 10
...
ICL> creframe
XDIM - x dimension of output array /10/ > 25
...
ICL> creframe reset
XDIM - x dimension of output array /64/ >
...
ICL>
```

When the CREFRAME program is executed for the first time, the suggested value of the `XDIM` parameter is '64'. In the first prompt above, the value of `XDIM` is set to '10'. When CREFRAME is executed for the second time, the suggested value of `XDIM` has changed to '10', i.e. it takes its 'current value' which was set during the previous execution of CREFRAME. In the third execution of CREFRAME we have included the keyword RESET in the command line. The effect is that when the value of parameter `XDIM` is prompted for, the suggested value has reverted to '64'.

RESET may be combined with the keywords PROMPT and ACCEPT described below.

PROMPT:

The PROMPT keyword forces a prompt to appear for every program parameter not specified on the command line. For example, program CREFRAME has a parameter called `TITLE` which normally takes the value 'KAPPA – Creframe' without prompting. However, if you use the keyword PROMPT in the command line, a prompt for TITLE will be made:

```
ICL> creframe prompt
...
TITLE - Title for output array /'KAPPA - Creframe'/ > 'My title'
ICL>
```

ACCEPT:

The ACCEPT keyword forces the parameter system to accept the suggested value for *every* program parameter. This must be used with care because some parameters may not have suggested values and need a value to be specified in order to work properly. For example, CREFRAME must have a value specified for parameter `OUTPIC`, the name of the output data object. If you ran the program like this:

```
ICL> creframe accept
```

the program would fail because it does not know where to write the output data. However, if you ran the program like this:

```
ICL> creframe outpic=ramp accept
```

The program would generate an output image using suggested values for all the parameters except `OUTPIC`, and write the output to file `RAMP.SDF`.

Sometimes the keyword ACCEPT can be used alone. For example, you could follow the above command by the command:

```
ICL> look accept
```

and the central `7x7` array of the image created by CREFRAME would be displayed on your terminal without any parameter values being prompted for.

The symbol '\' has the same effect as ACCEPT, thus:

```
ICL> look \
```

would have the same effect as the previous example — and is much quicker to type.

8.4 Value formats

When specifying parameter values it is important to remember whether you are talking to the ICL command interpreter or to the ADAM parameter system. The differences are highlighted in Section 8.5

8.4.1 ICL command parameters

Parameter value specifications in an ICL command can take three forms:

- An expression enclosed in parentheses. E.g.

```
(2)        (X)        (Y*SIN(THETA))
```

- A string enclosed in quotes. E.g.

```
'String'              "Filtered and calibrated"
```

- Any sequence of characters not including a space, quote, comma or left parenthesis. E.g.

```
NGC1635    X          PI*2
```

The first form is used to pass the value of an ICL expression to a command, or to give a command a variable in which to return a value. The other two forms both pass a string.

We can illustrate the three forms by using the PRINT command which prints its parameter values on the terminal:

```
ICL> X=1.234
ICL> PRINT (X)
1.234000
ICL> PRINT 'HELLO'
HELLO
ICL> PRINT HELLO
HELLO
```

In many cases you do not need to use the quoted form of a string because the simpler form will work. You need the quoted form for those cases in which you need to delimit a *single* string containing a left parenthesis or spaces. Consider the following example which contains a PRINT command which has several parameters:

```
ICL> X=2
ICL> PRINT The Square Root of (X) is (SQRT(X))
The Square Root of          2 is 1.414214
```

Since spaces are parameter separators, '`The`', '`Square`', '`Root`', and '`of`' are all received by PRINT as separate parameters. However, PRINT simply concatenates all its parameters with a space between each pair. Thus, the output written by PRINT is the string, just as you typed it. Many other ICL commands which accept strings work in this way. This means that strings with *single* spaces may not need quotes when used as command parameters. However, in general this is *not* true of commands which run ADAM programs when strings are specified on the command line (see below).

The rules for specifying ICL command parameters can be summarized as follows:

- To pass the value of an expression or the name of a variable, enclose it in parentheses.
- Anything not in parentheses is passed as a string, or as a sequence of strings if it contains spaces or commas.
- Any string which does not fit these restrictions can be passed by placing it in quotes. Quotes may be included in a string by typing two consecutive quotes.

8.4.2 ADAM program parameters

Values can be specified for ADAM program parameters in an ICL command line and in response to prompts. However, some values are only acceptable when given in response to prompts; they are meaningless to ICL.

Values acceptable in commands, and in response to prompts:

Numbers — These can be Integer, Real, or Double Precision, and are entered in the usual Fortran format. If necessary, the parameter system will perform the appropriate type conversion and truncation required to store the value in the form specified by the programmer.

Logicals — These can be represented by the words TRUE, FALSE, YES, NO, T, F, Y, N; regardless of case. ICL logical expressions can also be used, but only in commands.

Strings — These are strings of characters similar to the Fortran CHARACTER*n type. Strings can be enclosed in single (′) or double (″) quotes. Quotes may be omitted where there is no ambiguity (see below). A special case of a String is an object or logical *Name*. These need careful treatment and are discussed below.

Arrays — Arrays of any of the above types may be represented by a list of values enclosed in square brackets, e.g. [1, 2, 3]. Two dimensional arrays may be represented as [[1, 2], [3, 4]] etc.

Values acceptable only in response to prompts:

<**CR**> — Accept the suggested value.

<**TAB**> — The suggested value is put in the input buffer. You can then edit it using the normal keyboard editing facilities before entering the value in the normal way.

! — Return a STATUS value indicating 'Null Parameter Value'. This will often cause a program to abort, but it can force a suitable default value to be used when this is the most sensible action.

!! — Abort the program, by convention.

? — The parameter system will display the text specified in the 'help' field of the parameter specification in the program's interface file. If this field is missing, a blank line will be output. You will be reprompted for the parameter, for example:

```
MODE - Method for selecting contour heights /'FR'/ > ?
  Options are: FR = explicit list; AU = automatic selection;
  AR = equal-area; LI = linear; MA = magnitude
MODE - Method for selecting contour heights /'FR'/ >
```

?? — This puts you into an interactive Help session, providing access to information about more than just the parameter being prompted for. It is used like the ordinary VMS Help facility. When you exit (by pressing <CR> in response to a *Topic?* prompt) you will be returned to the original parameter prompt.

Strings:

You must be careful when specifying string values. A string which contains a space or comma, or which begins with a left parenthesis, should be enclosed in single or double quotes. For instance, if you specify the string 'Sum of 2 images' in an ICL command, you could type:

```
ICL> TESTC 'Sum of 2 images'
```

in which case the parameter system would be handed the string ‘Sum of 2 images’ as the value of the first parameter of TESTC. However, if you type:

```
ICL> TESTC Sum of 2 images
```

the parameter system will think there are four parameters ‘`Sum`’, ‘`of`’, ‘`2`’, and ‘`images`’.

Quotes are not needed when a string is input in response to a prompt. For example:

```
ICL> TESTC
X - x value/'Default'/ > Sum of 2 images
```

will successfully give parameter X the value ‘Sum of 2 images’ (provided X is of type _CHAR or LITERAL).

The use of single quotes (’) is dangerous when used from DCL as DCL will interpret them as indicating symbol substitution. Thus, when your input will be processed by the DCL command interpreter, double quotes (") should be used.

Object names:

Strings are used to specify the names of files, data objects, and devices (graphics devices, tape drives etc.). Components of data objects are specified by a dot notation, e.g. RAMP1.TITLE refers to the TITLE component of the image RAMP1.

Normally, a name can be typed without quotes. However, ambiguities can occur: a file name such as [ABC.DEF]FILE begins with a ‘[’ character and may be confused with an array specification. Such ambiguities can be resolved by prefixing the name with an ‘@’ character, e.g. @[ABC.DEF]FILE means the file of this name. As another example, in the prompt line:

```
PLTITL - Plot title /' '/ > @ADAM_USER:GALAXY.MYTITLE
```

the parameter PLTITL is given the value of the string contained in the object MYTITLE in `GALAXY.SDF`. If the ‘`@`’ were omitted, the string ‘`ADAM_USER:GALAXY.MYTITLE`’ would be used as the value of PLTITL. Note that the file extension (.SDF) should not be included when giving the name of a data file, otherwise the data system will look for the component SDF within the object GALAXY.MYTITLE.

Logical names:

Logical names must be defined with the /JOB qualifier. Thus, if your IMAGE data files are stored in IMAGEDIR alias `DISK$USER1:[XYZ.IRCAM.IMAGES]`:

```
$ DEFINE/JOB IMAGEDIR DISK$USER1:[XYZ.IRCAM.IMAGES]
```

will enable you to respond to a prompt thus:

```
INPIC - Input image /@ramp4/ > imagedir:ngc1365
```

8.5 Comparison of ICL and ADAM parameters

Some values which are acceptable when specified as a parameter value in an ICL command are *not* acceptable when used in response to prompts from the ADAM parameter system. These unacceptable values are ICL functions. Thus, suppose program TESTR has been written to read a real number and print it out. We could do the following:

```
ICL> DEFINE TESTR TESTR
ICL> TESTR (TAND(60))
ICL> TESTR prints 1.732051
ICL>
```

However, an attempt to use the ICL function TAND in a response to a prompt would be rejected:

```
ICL> TESTR
X - x value/1.732051/ > (TAND(60))
%RMS-F-SYN, file specification syntax error
X - x value/1.732051/ >
```

You will have to specify a value which is acceptable to the ADAM parameter system.

Chapter 11 shows some example ADAM programs which read a parameter value and display it on the user's terminal. These can be used to examine how ICL and the ADAM parameter system process input values. Program TESTI reads a parameter value of type '_INTEGER'. Here is how this program responds to various inputs specified first on the command line and then in response to a prompt:

Input	Output	
	Command line	Prompt
5	5	5
5.6	6	6
-1234	-1234	-1234
tand(60)	File not found	File not found
(tand(60))	2	file specification syntax error
(%B100110)	38	file specification syntax error

In this table the first column shows the input value, the second shows the value output by the program when the input value is given on the command line, and the third shows the value output when the input is given in response to a prompt. Thus, the second and third columns show the parameter value presented to the program, or the error status if no legal value could be obtained.

Notice that:

- The input value '5.6' is rounded up to '6' by the parameter system before being given to the program. This is because the value is obtained by calling routine PAR_GET0I which converts an input value to integer type.
- The ICL function 'tand(60)' must be enclosed in parentheses if it is to be recognised as an expression, otherwise it is interpreted as an object name and the system goes looking for its container file.
- The expressions '(tand(60))' and '(%B100110)' are recognised as expressions when input on the command line, but not when input in response to prompts. When it is recognised, the value of tand(60) is rounded up from 1.732051 to 2.

The next table shows the response of program TESTR which reads the value of a parameter of type '_REAL' from the environment and displays this value on the terminal:

Input	Output	
	Command line	Prompt
5	5	5
5.6	5.6	5.6
1.23D12	1.23E12	1.23E12
(tand(60))	1.732051	file specification syntax error
true	File not found	File not found
'true'	Conversion error	Conversion error

Notice that in this case the input values 5.6 and (tand(60)) are not rounded to integers as the program is expecting real numbers.

The next table shows the behaviour of program TESTL which is like the previous programs but whose parameter is of type '_LOGICAL':

Input	Output	
	Command line	Prompt
5	Conversion error	Conversion error
TRUE	TRUE	TRUE
T	TRUE	TRUE
n	FALSE	FALSE
(lge(a,b))	TRUE	File not found

Acceptable inputs for logical values are:

```
TRUE  true  YES  yes    T    t    Y    y
FALSE false NO   no     F    f    N    n
```

Notice that the ICL function 'lge' is acceptable when specified on the command line, but not in response to a prompt.

The final table shows the input to and output from program TESTC, which is the same as the previous programs but has a parameter value of type '_CHAR':

Input	Output	
	Command line	Prompt
5.6	5.6	5.6
tand(60)	tand(60)	tand(60)
(tand(60))	1.73205080	(tand(60))
yogi bear	yogi	yogi bear
'yogi bear'	yogi bear	yogi bear
"yogi bear"	yogi bear	yogi bear
(yogi bear)	Right parenthesis expected	(yogi bear

Notice that:

- For 'command line' input, ICL expressions are *evaluated* and the result is converted into a string before being given to the program. For 'prompt' input, the expressions are not evaluated but are just regarded as strings of characters.
- Spaces in strings separate parameters in 'Command line' input when the string is not enclosed in quotes or double quotes.

If you read Chapter 11 you will be able to create your own versions of the TESTx programs and try out various inputs on the system yourself.

Chapter 9
Programming in ICL

9.1 Control statements

These statements provide a means of defining ICL procedures and of controlling the flow of execution within them. There are four of them:

> **PROC** — Defines a procedure.
> **IF** — Provides a decision-making structure.
> **LOOP** — Provides a looping structure.
> **EXCEPTION** — Provides an error-handling structure.

Unlike direct statements, they can only be used in a procedure and are not accepted in direct mode.

9.1.1 IF

The IF statement is essentially the same as the block IF of Fortran. It has the following general form:

```
IF expression
    statements
ELSE IF expression
    statements
ELSE IF expression
    statements
...
ELSE
    statements
END IF
```

The expressions (which must give logical values) are evaluated in turn until one is found to be true; the following statements are then executed. If none of the expressions are true, the statements following ELSE are executed.

Every IF statement must begin with IF and end with END IF (or ENDIF). The ELSE IF (or ELSEIF) and ELSE clauses are optional, so the simplest IF statement would have the form:

```
IF expression
    statements
ENDIF
```

The following example procedure illustrates the use of the IF statement, and shows how they may be nested inside each other:

```
PROC QUADRATIC A,B,C
{ A Procedure to find the roots of the quadratic equation
{ A*X**2 + B*X + C = 0
   IF A=0 AND B=0
      PRINT The equation is degenerate
   ELSE IF A=0
      PRINT Single Root is (-C/B)
   ELSE IF C=0
      PRINT The roots are (-B/A) and 0
   ELSE
      RE = -B/(2*A)
      DISCRIMINANT = B*B - 4*A*C
      IM = SQRT(ABS(DISCRIMINANT)) / (2*A)
      IF DISCRIMINANT >= 0
         PRINT The Roots are (RE + IM) and (RE - IM)
      ELSE
         PRINT The Roots are complex
         PRINT (RE) +I* (IM) and
         PRINT (RE) -I* (IM)
      ENDIF
   ENDIF
END PROC
```

9.1.2 LOOP

The LOOP statement is used to execute repeatedly a group of statements. It has three different forms:

- LOOP
- LOOP FOR
- LOOP WHILE

LOOP:

This is the simplest form of looping structure:

```
LOOP
    statements
END LOOP
```

This form sets up an infinite loop. Fortunately, an additional statement (BREAK) can be used to terminate the loop; BREAK would normally appear inside an IF statement within the loop. For example:

```
PROC COUNT
{ A procedure to print the numbers from 1 to 10
   I = 1
   LOOP
      PRINT (I)
      I = I+1
      IF I>10
         BREAK
      ENDIF
   ENDLOOP
ENDPROC
```

LOOP FOR:

As it is frequently required to loop over a sequential range of numbers, a special form of the LOOP statement is provided for this purpose. It has the following form:

```
LOOP FOR variable = expression1 TO expression2 [STEP expression3]
    statements
END LOOP
```

This form is essentially equivalent to the DO loop in Fortran. The expressions specifying the range of values for the control variable are rounded to the nearest integer so that the variable always has an integer value. Using this form of the LOOP statement you can simplify the previous example as follows:

```
PROC COUNT
{ A procedure to print the numbers from 1 to 10
   LOOP FOR I = 1 TO 10
      PRINT (I)
   ENDLOOP
ENDPROC
```

Note that there is an optional STEP clause in the LOOP FOR statement. If this is not specified, a STEP of 1 is assumed. The STEP clause can be used to specify a different value. A step of -1 must be specified to get a loop which counts down from a high value to a lower value. For example:

```
LOOP FOR I = 10 TO 1 STEP -1
```

will count down from 10 to 1.

LOOP WHILE:

The third form of LOOP statement specifies loops which terminate on a condition. It has the form:

```
LOOP WHILE expression
    statements
END LOOP
```

The expression is evaluated each time round the loop and if it has the logical value TRUE, the statements which form the body of the loop are executed. If it has the value FALSE, execution continues with the statement following END LOOP. Using this form you can write yet another version of the COUNT procedure:

```
PROC COUNT
{ A procedure to print the numbers from 1 to 10
   I = 1
   LOOP WHILE I<=10
      PRINT (I)
      I = I+1
   ENDLOOP
ENDPROC
```

In the above case, the LOOP WHILE form is more complicated than the LOOP FOR form. However, LOOP WHILE can be used to express more general forms of loop where the termination condition is something derived inside the loop. An example is a program which prompts you for an answer to a question and has to keep repeating the prompt until a valid answer is received:

```
FINISHED = FALSE
LOOP WHILE NOT FINISHED
   INPUT Enter YES or NO:  (ANSWER)
   FINISHED = ANSWER = 'YES' OR ANSWER = 'NO'
END LOOP
```

9.2 Procedures and command files

LIST	List a procedure
EDIT	Edit a procedure
SET EDITOR	Change editor used by EDIT
SAVE	Save a procedure
LOAD	Accept commands from a saved procedure
DELETE	Delete a procedure
PROCS	List procedure names
VARS	List procedure variables
SET (NO)TRACE	Switch tracing of procedures on/off

An ICL procedure is like a subroutine in Fortran. It allows you to write a sequence of ICL statements which can later be run with a single command. The procedure may have parameters which are used to pass values to the procedure and return values from it.

9.2.1 Defining procedures — PROC

To define a procedure, type a PROC command which specifies the name of the procedure and the names of its parameters. ICL then returns a new prompt using the name of the procedure (rather than `ICL>`) to show that you are in the procedure entry phase of procedure mode. The statements that make up the procedure are then entered, followed by an `END PROC` or `ENDPROC` to mark the end of the procedure. For example:

```
ICL> PROC SQUARE_ROOT X
SQUARE_ROOT> { An ICL procedure to print the square root of a number
SQUARE_ROOT> PRINT The Square Root of (X) is (SQRT(X))
SQUARE_ROOT> END PROC
ICL>
```

9.2.2 Running procedures

To run the procedure you have entered, use its name as an ICL command and add any parameter values required:

```
ICL> SQUARE_ROOT (2)
The Square Root of         2 is 1.414214
```

9.2.3 Listing procedures — LIST

The LIST command lists a procedure on the terminal. Just type 'LIST', followed by the name of the procedure you want to list:

```
ICL> LIST SQUARE_ROOT

    PROC SQUARE_ROOT X
```

```
        { An ICL procedure to print the square root of a number
        PRINT The Square Root of (X) is (SQRT(X))
        END PROC

    ICL>
```

9.2.4 Editing procedures — EDIT, SET EDITOR

Typing in procedures directly is fine for very simple procedures, but for anything complex it is likely that some mistakes will be made. When this happens, it will be necessary to *edit* the procedure. Editing can be done from within ICL using standard editors. For example the command:

```
    ICL> EDIT SQUARE_ROOT
```

can be used to edit the SQUARE_ROOT procedure. By default the TPU editor is used. It is also possible to select the EDT or LSE editors using the SET EDITOR command. For example, to select EDT, type:

```
    ICL> SET EDITOR EDT
```

When editing a procedure there are two possible options:

- You can leave the name of the procedure (specified in the PROC statement) unchanged, but edit the code. This creates a new version of the procedure which replaces the old one *when you exit from the editing session* — in other words, the old version will still be used until you terminate the edit.
- You can change the name of the procedure by editing the PROC statement. This creates a new procedure with the new name, and leaves the old procedure unchanged.

It is possible to create procedures from scratch using the editors. However, it is recommended that procedures be typed in directly to start with. The advantage is that during direct entry, any syntactic errors will be detected immediately. Thus, if you mistype the PRINT line in the above example you get an error message as follows:

```
    SQUARE_ROOT> PRINT The Square Root of (X is (SQRT(X))
    PRINT The Square Root of (X is (SQRT(X))
                                ^
    Right parenthesis expected
```

The error message consists of the line in which the error was detected, a pointer which indicates where in the line ICL had got to when it found something was wrong, and a message indicating what was wrong. In this case it encountered the `'is'` string when a right parenthesis was expected. Following such an error message, you can use the command line editing facility to correct the line and reenter it. If the same error occurred during procedure entry using an editor, the error message would only be generated at the time of exit from the editing session, and it would be necessary to edit the procedure again to correct it.

9.2.5 Direct execution of statements during procedure entry

It is sometimes useful to have a statement executed directly while entering a procedure. When using the direct entry method, this can be done by prefixing the command with a '%' character. For example:

```
    ICL> PROC SQUARE_ROOT X
    SQUARE_ROOT> %HELP
```

would give you on-line help information you might need to complete the procedure. If the '%' was omitted, the HELP command would be included as part of the procedure.

9.2.6 Saving, loading and deleting procedures — SAVE, LOAD, DELETE

Procedures created by direct entry will only exist for the duration of an ICL session. If you need to keep them longer, they need to be saved in a disk file. This is achieved by means of the SAVE command. To save your SQUARE_ROOT procedure on disk you would use:

```
ICL> SAVE SQUARE_ROOT
```

To load it again, probably in a subsequent ICL session, you would use the LOAD command.

```
ICL> LOAD SQUARE_ROOT
```

The SAVE command causes the procedure to be saved in a file with name SQUARE_ROOT.ICL in the current default directory. LOAD will load the procedure from the same file, also in the default directory. However, with LOAD it is possible to specify an alternative directory if required:

```
ICL> LOAD DISK$USER:[ABC]SQUARE_ROOT
```

If you have many procedures, you may not want to save and load them all individually. It is possible to save *all* the current procedures using the command:

```
ICL> SAVE ALL
```

This saves them in a single file with the name SAVE.ICL. They may then be reloaded with the command:

```
ICL> LOAD SAVE
```

The SAVE ALL command is rarely used, however, because it is executed automatically when you exit from ICL. This ensures that ICL procedures do not get accidentally lost because you forget to save them.

Procedures can be deleted with the DELETE command. Just follow the command name with the name of the procedure to be deleted:

```
DELETE  name
```

9.2.7 Variables

Any variable used within a procedure is completely distinct from a variable of the same name used outside the procedure or within a different procedure, as can be seen in the following example:

```
ICL> X=1
ICL> PROC FRED
FRED> X=1.2345
FRED> =X
FRED> END PROC
ICL> FRED
1.234500
ICL> =X
        1
ICL>
```

When you run the procedure FRED you get the value of the variable X within the procedure. Then, typing '=X' gives the value of X outside the procedure, which has remained unchanged during execution of the procedure. This feature has the consequence that you can use procedures freely without having to worry about any possible side effects on variables outside them.

The situation is exactly the same as that in Fortran where variables in a subroutine are local to the subroutine in which they are used. In Fortran, the COMMON statement is provided for use in cases where it is required to extend the scope of a variable over more than one routine. ICL does not have a COMMON facility, but does provide an alternative mechanism for accessing variables outside their scope using the command VARS and the function VARIABLE.

9.2.8 Finding out what is available — VARS, PROCS

The VARS command lists all the variables of a procedure. It has one parameter, which is the name of the procedure. If the parameter is omitted, the outer level variables, i.e. those that are not part of any procedure, are listed. Thus, after the previous example you would get:

```
ICL> VARS FRED
             X  REAL     1.2345000000000E+00
ICL> VARS
             X  INTEGER          1
ICL>
```

VARIABLE():

The function VARIABLE gives the value of a specified variable in a specified procedure, for example:

```
ICL> =VARIABLE(FRED,X)
1.234500
ICL>
```

and thus allows a variable belonging to a procedure to be accessed outside that procedure.

Note that the variables belonging to a procedure continue to exist after it finishes execution, and if the procedure is executed a second time they will retain their values from the first time through the procedure on entry to the procedure for the second time.

PROCS:

To find the names of all the current procedures, use the 'PROCS' command:

```
ICL> PROCS

SQUARE_ROOT

ICL>
```

9.2.9 Tracing execution — SET (NO)TRACE

The commands SET TRACE and SET NOTRACE switch ICL in and out of trace mode. When in trace mode, each statement executed will be listed on the terminal. Trace mode is very useful for debugging procedures. The commands can be issued either from direct mode to turn on tracing for the entire execution of a procedure, or inserted in the procedure itself, making it possible to trace just part of its execution.

9.2.10 Command files

You have seen that the LOAD command 'loads' a procedure that had previously been stored in a file by a SAVE command. What

```
ICL> LOAD SQUARE_ROOT
```

actually does is to read ICL commands from the file SQUARE_ROOT.ICL *and obey them*. In the case of SQUARE_ROOT, these commands define a procedure of the same name. However, there is nothing to stop us storing any ICL commands you like in a file of type .ICL, and then you can load and execute them using the LOAD command. The PROC statement is designed only for creating procedures, so for creating general .ICL files you must use a normal editor. These general .ICL files are called *Command Files*.

A common use of command files is to store a set of command definitions for a monolith. You have already seen an example of this in file KAPPA_DIR:KAPPA.PRC. When you startup the KAPPA package, this command:

```
ICL> LOAD KAPPADIR:KAPPA.PRC
```

is automatically executed, and this causes the command definitions it contains to become effective. An example of setting up your own command file is given in Section 11.8.

A subtlety to be aware of is the difference between storing commands within a procedure in a command file, and just storing the commands. For example, if you wanted to store the command definitions for your TESTx programs (see Chapter 11) in a procedure (given that these programs had all been put into a monolith called TEST), you could do this from within ICL as follows:

```
ICL> PROC TEST
TEST> { Define commands to run the TESTx programs
TEST> DEFINE TESTC TEST
TEST> DEFINE TESTI TEST
TEST> DEFINE TESTL TEST
TEST> DEFINE TESTR TEST
TEST> END PROC
ICL> SAVE TEST
```

In a later ICL session you could load this procedure from the file TEST.ICL in which it had been stored:

```
ICL> LOAD TEST
```

This command would read the commands from TEST.ICL and obey them, and this would have the effect of defining the procedure TEST. *However, the DEFINE commands within the procedure would not be executed until the procedure was executed.* Thus, the commands TESTC etc would not be recognised until you had run the procedure as follows:

```
ICL> TEST
```

However, if you used your favourite editor to store the following ICL commands in the command file TEST.ICL (independently of any ICL session):

```
{ Define commands to run the TESTx programs
DEFINE TESTC TEST
DEFINE TESTI TEST
DEFINE TESTL TEST
DEFINE TESTR TEST
```

in your next ICL session you could execute these commands by typing:

```
ICL> LOAD TEST
```

The difference from the previous method is that the commands TESTC etc are now defined and can be used immediately without having to execute a procedure.

9.2.11 Running ICL as a batch job

It is sometimes useful to run one or more ICL procedures as a batch job in VMS. It is quite easy to set this up by using a parameter with the ICL command to specify a file from which commands will be taken:

```
$ ICL filename
```

This form of the command is equivalent to typing ICL and then typing:

```
ICL> LOAD filename
```

Note that a LOAD file may include direct commands as well as procedures. In order to create a Batch job, you must set up a file which contains all the procedures wanted, a command (or commands) to run them, and an EXIT command to terminate the job. Here is the file for a simple Batch job to print a table of square roots using your earlier example procedure:

```
PROC SQUARE_ROOT X
{ An ICL procedure to print the square root of a number
   PRINT The Square Root of (X) is (SQRT(X))
END PROC

PROC TABLE
{ A procedure to print a table of square roots of numbers from 1 to 100
   LOOP FOR I=1 TO 100
      SQUARE_ROOT (I)
   END LOOP
END PROC

{ Next, the command to run this procedure

TABLE

{ And then an EXIT command to terminate the job

EXIT
```

This file can be generated using the EDIT command from DCL. If the procedures have already been tested from ICL, it is convenient to use a SAVE ALL command (or exit from ICL) to save them, and then edit the SAVE.ICL file to add the additional direct commands. Suppose this file is called TABLE.ICL, then to create a batch job, a command file is needed which could be called TABLE.COM and would contain the following:

```
$ ICL TABLE
$ EXIT
```

It might also need to contain a SET DEF command to set the appropriate directory, or a directory specification on the TABLE file name if it is not in the top level directory.

To submit the job to the batch queue, the following command is used:

```
$ SUBMIT/KEEP TABLE
```

The /KEEP qualifier specifies that the output file for the batch job is to be kept. This file will appear as TABLE.LOG in your top level directory and will contain the output from the batch job. An /OUTPUT qualifier can be used to specify a different file name or directory for it.

9.3 Exceptions

SIGNAL	Signal an ICL exception

Error conditions and other unexpected events are referred to as *Exceptions*. When such a condition is detected in direct mode, a message is output. For example, if you enter a statement which results in an error:

```
ICL> =SQRT(-1)
SQUROONEG   Square Root of Negative Number
ICL>
```

you get a message consisting of the name of the exception (SQUROONEG) and a description of the nature of the exception. A full list of ICL exceptions is given in Chapter 19.

If the error occurs within a procedure, the message contains a little more information. For example, if you use your square root procedure with an invalid value, you get the following messages:

```
ICL> SQUARE_ROOT (-1)
SQUROONEG   Square Root of Negative Number
In Procedure: SQUARE_ROOT
At Statement: PRINT  The Square Root of (X) is (SQRT(X))
ICL>
```

If one procedure is called by another, the second procedure will also be listed in the error message. For example, if you run the following procedure:

```
PROC TABLE
{ Print a table of Square roots from 2 down to -2
   LOOP FOR I = 2 TO -2 STEP -1
     SQUARE_ROOT (I)
   END LOOP
END PROC
```

you get:

```
ICL> TABLE
The Square Root of 2 is 1.414214
The Square Root of 1 is 1
The Square Root of 0 is 0
SQUROONEG   Square Root of Negative Number
In Procedure: SQUARE_ROOT
At Statement: PRINT  The Square Root of (X) is (SQRT(X))
Called by: TABLE
ICL>
```

9.3.1 Exception handlers

It is often useful to be able to modify the default behaviour on an error condition. You may not want to output an error message and return to the ICL> prompt, but rather to handle the condition in some other way. This can be done by writing an *Exception Handler*. Here is an example of an exception handler in the SQUARE_ROOT procedure:

```
PROC SQUARE_ROOT X
{ An ICL procedure to print the square root of a number
   PRINT The Square Root of (X) is (SQRT(X))
   EXCEPTION SQUROONEG
{ Handle the imaginary case
      SQ = SQRT(ABS(X))
      PRINT The Square Root of (X) is (SQ&'i')
   END EXCEPTION
END PROC
```

Now running the TABLE procedure gives:

```
ICL> TABLE
The Square Root of 2 is 1.414214
The Square Root of 1 is 1
The Square Root of 0 is 0
The Square Root of -1 is 1i
The Square Root of -2 is 1.414214i
ICL>
```

The exception handler has two effects. First, the code contained in the exception handler is executed when the exception occurs. Second, the procedure exits normally to its caller (in this case TABLE) rather than aborting execution completely and returning to the ICL> prompt.

Exception handlers should be placed after the normal code, but before the END PROC statement. There may be any number of exception handlers in a procedure, each for a different exception. The exception handler begins with an EXCEPTION statement specifying the exception name, and finishes with an END EXCEPTION statement. Between these may be any ICL statements, including calls to other procedures.

An exception handler does not have to be in the procedure causing the exception, but could be in a procedure further up the chain of calls. In your example you could put an exception handler for SQUROONEG in TABLE rather than in SQUARE_ROOT:

```
PROC TABLE
{ Print a table of Square roots from 2 down to -2
   LOOP FOR I = 2 TO -2 STEP -1
      SQUARE_ROOT (I)
   END LOOP
   EXCEPTION SQUROONEG
      PRINT 'Can''t handle negative numbers - TABLE Aborting'
   END EXCEPTION
END PROC
```

giving:

```
ICL> TABLE
The Square Root of 2 is 1.414214
The Square Root of 1 is 1
The Square Root of 0 is 0
Can't handle negative numbers - TABLE aborting
ICL>
```

Below is an example of a pair of procedures which use an exception handler for floating point overflow in order to locate the largest floating point number allowed on the system. Starting with a value of 1, this is multiplied by 10 repeatedly until floating point overflow occurs. The highest value found in this way is then multiplied by 1.1 repeatedly until overflow occurs, then by 1.01 etc:

```
PROC LARGE  START, FAC, L
{ Return in L the largest floating point number before
{ overflow occurs when START is repeatedly multiplied by FAC.
   L = START
   LOOP
      L = L * FAC
   END LOOP
   EXCEPTION FLTOVF
{ This exception handler doesn't have any code - it just
{ causes the procedure to exit normally on overflow.
   END EXCEPTION
END PROC

PROC LARGEST
{ A Procedure to find the largest allowed floating point number on the system.
   FAC = 10.0
   LARGE  1.0, (FAC), (L)
   LOOP WHILE FAC > 0.00000001
      LARGE (L), (1.0+FAC), (L)
      FAC = FAC/10.0
   END LOOP
   PRINT  The largest floating point number allowed is (L)
END PROC
```

9.3.2 Keyboard aborts

One exception which is commonly encountered is that which results when a ctrl/C is entered on the terminal. This results in the exception CTRLC and may therefore be used to abort execution of a procedure and return ICL to direct mode. However, an exception handler for CTRLC may be added to a procedure to modify the behaviour when a ctrl/C is typed.

9.3.3 SIGNAL

The exceptions described up to now have all been generated internally by the ICL system, or, in the case of CTRLC, initiated by the user. It is also possible for ICL procedures to generate exceptions which may be used to indicate error conditions. This is done by the SIGNAL command which has the form:

```
SIGNAL  name  text
```

where `name` is the name of the exception, and `text` is the message text associated with the exception. The exception name may be any valid ICL identifier. Exceptions generated by SIGNAL work in exactly the same way as the standard exceptions listed in SG/5. An exception handler will be executed if one exists, otherwise an error message will be output and ICL will return to direct mode.

One use of the SIGNAL command is as a means of escaping from deeply nested loops. The BREAK statement can be used to exit from a single loop, but is not applicable if two or more loops are nested. In these cases, the following structure could be used:

```
LOOP
   LOOP
```

```
      LOOP
         ...
         IF FINISHED
            SIGNAL ESCAPE
         END IF
         ...
      END LOOP
   END LOOP
END LOOP

EXCEPTION ESCAPE
END EXCEPTION
```

where the exception handler again contains no statements, but simply exists to cause normal procedure exit, rather than an error message, when the exception is signalled.

9.4 ICL login files

If you frequently need to define commands to run particular programs, it is convenient to define them in an *ICL login file* which will be loaded automatically each time ICL is started up. An ICL login file works in exactly the same way as a DCL login file. ICL uses the logical name ICL_LOGIN to locate this file, so store the required definitions in a file called LOGIN.ICL in your top level directory, and put the following definition in your DCL LOGIN.COM file:

```
$ DEFINE ICL_LOGIN DISK$USER:[ABC]LOGIN.ICL
```

where DISK$USER:[ABC] needs to be replaced by the actual directory used. This command file will then be loaded automatically whenever you start up ICL and can include procedures, definitions of commands, or indeed any valid ICL command. Below is an example of an ICL login file which illustrates some of the facilities which may be used:

```
{ ICL Login File

{ Define TYPE command
DEFSTRING  T(YPE)  DCL TYPE

{ Define EDIT command
HIDDEN PROC EDIT name
   IF INDEX(name,'.') = 0
      #EDIT (name)
   ELSE
      $ EDIT (name)
   ENDIF
END PROC

{ Login Message
PRINT
PRINT   Starting ICL at (TIME()) on (DATE())
PRINT
```

9.4.1 Hidden procedures

The definition of the EDIT command shown in the login file above is done using a *Hidden procedure*. Since EDIT is an ICL command to edit procedures, if you just used DEFSTRING to define EDIT as DCL EDIT, you would lose the ability to edit ICL procedures — the EDIT command would always edit VMS files. The procedure used to redefine EDIT gets around this by testing for the existence of a dot in the name of the file to be edited using the INDEX function. If a dot is present, it assumes that a VMS file is being edited and issues the command '$ EDIT (name)'. If no dot is present, it assumes that an ICL procedure is being edited and the command '#EDIT (name)' is issued. The # character forces the internal definition of EDIT to be used, rather than the definition currently being defined.

The procedure is written as a *hidden* procedure, indicated by the word HIDDEN preceding PROC. A hidden procedure works in exactly the same way as a normal procedure, but it does not appear in the listing of procedures produced by a PROCS statement, nor can it be edited, deleted, or saved from within ICL. It is convenient to make all procedures in your login file hidden procedures so that they do not clutter your directory of procedures and cannot be deleted accidentally.

9.5 Extending on-line help

HELP	Display on-line documentation
DEFHELP	Define the source of Help information

ICL includes a HELP command which provides on-line documentation on ICL itself. Using the DEFHELP command it is possible to extend this facility to access information on the commands you have added. In order to do this, you need to create a help library in the normal format used by the VMS help system. This is described in the VAX/VMS documentation for the Librarian utility. You can then specify topics from this library which will be available using the ICL HELP command by using a command of the form:

```
DEFHELP EDIT LIBRARY.HLB
```

This will cause a:

```
HELP EDIT
```

command to return the information on EDIT in help library LIBRARY.HLB, rather than in the standard ICL library.

9.6 Example procedures

This final section lists some examples of ICL procedures taken from the KAPPA manual (SUN/95). They should help you understand how to use the programming facilities of ICL correctly. Many of the commands used in the procedures are KAPPA applications.

Unsharpmask:

Suppose you have a series of commands to run on a number of files. You could create a procedure to perform all the stages of the processing, deleting the intermediate files that it creates.

```
PROC UNSHARPMASK NDFIN CLIP NDFOUT

{ Clip the image to remove the cores of stars and galaxies above
{ a nominated threshold.
   THRESH (NDFIN) TMP1 THRHI=(CLIP) NEWHI=(CLIP) \

{ Apply a couple of block smoothings with boxsizes of 5 and 13
{ pixels.  Delete the temporary files as we go along.
   BLOCK TMP1 TMP2 5
   $ DELETE TMP1.SDF;0
   BLOCK TMP2 TMP3 13
   $ DELETE TMP2.SDF;0

{ Multiply the smoothed image by a scalar.
   CMULT TMP3 0.8 TMP4
   $ DELETE TMP3.SDF;0

{ Subtract the smoothed and renormalised image from the input image.
{ The effect is to highlight the fine detail, but still retain some of the
{ low-frequency features.
   SUB (NDFIN) TMP4 (NDFOUT)
   $ DELETE TMP4.SDF;0
END PROC
```

Multistat:

A common use of procedures is likely to be duplicate processing for several files. Here is an example procedure that does that. It uses some intrinsic functions which look just like Fortran.

```
PROC MULTISTAT

{ Prompt for the number of NDFs to analyse.  Ensure that it is positive.
   INPUTI Number of frames:  (NUM)
   NUM = MAX(1, NUM)

{ Find the number of characters required to format the number as
{ a string using a couple of ICL functions.
   NC = INT(LOG10(NUM)) + 1

{ Loop NUM times.
   LOOP FOR I=1 TO (NUM)

{ Generate the name of the NDF to be analysed via the ICL function SNAME.
     FILE = '@' & SNAME('REDX',I,NC)

{ Form the statistics of the image.
     STATS NDF=(FILE)
   END LOOP
END PROC
```

If NUM is set to 10, the above procedure obtains the statistics of the images named REDX1, REDX2, . . . REDX10. The ICL variable FILE is in parentheses because its value is to be substituted into parameter NDF. There is a piece of syntax to note which often catches people out. Filenames passed via ICL variables, such as FILE in the above example, must be preceded by an `@`.

Flatfield:

Here is another example, which could be used to flat field a series of CCD frames. Instead of executing a specific number of files, you can enter an arbitrary sequence of NDFs. When processing is completed a !! is entered rather than an NDF name, and that exits the loop. Note the ˜ continuation character. (It's not required but it's included for pedagogical reasons.)

```
PROC FLATFIELD

{ Obtain the name of the flat-field NDF.  If it does not have a
{ leading @ insert one.
   INPUT Which flat field frame?: (FF)
   IF SUBSTR(FF,1,1) <> '@'
      FF = '@' & (FF)
   END IF

{ Loop until there are no further NDFs to flat field.
   MOREDATA = TRUE
   LOOP WHILE MOREDATA

{ Obtain the frame to flat field.  Assume that it will not have
{ an @ prefix. Generate a title for the flattened frame.
     INPUT Enter frame to flat field (!! to exit): (IMAGE)
      MOREDATA = IMAGE = '!!'
      IF MOREDATA
         TITLE = 'Flat field of ' & (IMAGE)
         IMAGE = '@' & (IMAGE)

 { Generate the name of the flattened NDF.
         IMAGEOUT = (IMAGE) & 'F'
         PRINT Writing to (IMAGEOUT)

 { Divide the image by the flat field.

         DIV IN1=(IMAGE) IN2=(FF) OUT=(IMAGEOUT) ~
             OTITLE= (TITLE)
      ENDIF
   END LOOP
END PROC
```

Colstar:

Some KAPPA applications, particularly the statistical ones, produce output parameters which can be passed between applications via ICL variables. Here is an example to draw a perspective histogram centred about a star in a nominated data array from only the star's approximate position. The region about the star is stored in an output NDF file. Note, in a procedure meant to be used in earnest, there would be checks that input and output names begin with an @.

```
PROC COLSTAR FILE,X,Y,SIZE,OUTFILE

{+
{  Arguments:
{     FILE = FILENAME (Given)
{        Input NDF containing one or more star images.
{     X = REAL (Given)
{        The approximate x position of the star.
{     Y = REAL (Given)
{        The approximate y position of the star.
{     SIZE = REAL (Given)
{        The half-width of the region about the star's centroid to be
{        plotted and saved in the output file.
{     OUTFILE = FILENAME (Given)
{        Output primitive NDF of 2*%SIZE+1 pixels square (unless
{        constrained by the size of the data array or because the location
{        of the star is near an edge of the data array.
{-
```

```
{ Search for the star in a 21x21 pixel box.  The centroid of the
{ star is stored in the ICL variables XC and YC.
   CENTROID INPIC=(FILE) XINIT=(X) YINIT=(Y) XCEN=(XC) YCEN=(YC) ~
     MODE=INTERFACE SEARCH=21 MAXSHIFT=14

{ Convert the co-ordinates to pixel indices.
   IX = NINT(XC + 0.5)
   IY = NINT(YC + 0.5)

{ Find the upper and lower bounds of the data array to plot. Note
{ this assumes no origin information is stored in the data file.
   XL = MAX(1, IX - SIZE)
   YL = MAX(1, IY - SIZE)
   XU = MAX(1, IX + SIZE)
   YU = MAX(1, IY + SIZE)

{ Create a new IMAGE file centred on the star.
   PICK2D INPIC=(FILE) OUTPIC=(OUTFILE) XSTART=(XL) YSTART=(YL) ~
      XFINISH=(XU) YFINISH=(YU)

{ Draw a perspective histogram around the star on the current
{ graphics device.
   COLUMNAR IN=(OUTFILE)

{ Exit if an error occurred, such as not being to find a star
{ near the supplied position, or being unable to make the plot.
   EXCEPTION ADAMERR
      PRINT Unable to find or plot the star.
   END EXCEPTION
END PROC
```

Fancylook:

This creates a fancy display of an image with axes and a key showing data values. Note the need to give an expression combining the *x*-*y* bounds of the key to the LBOUND and UBOUND parameter arrays.

```
PROC FANCYLOOK NDF

{ Find the extent of the current picture.
   GDSTATE NCX1=(FX1) NCX2=(FX2) NCY1=(FY1) NCY2=(FY2) NOREPORT

{ Display the image with axes using the most-ornate font.
   DISPLAY (NDF) MODE=PE AXES FONT=NCAR COSYS=D SCALOW=(LOW) SCAHIGH=(HIGH) \

{ Find the extent of the image picture.
   PICIN NCX1=(DX1) NCX2=(DX2) NCY1=(DY1) NCY2=(DY2) NOREPORT

{ Determine the widths of the borders.
   XL = DX1 - FX1
   XR = FX2 - DX2
   YB = DY1 - FY1
   YT = FY2 - DY2

{ Only plot a key if there is room.
   IF MAX(XL, XR, YB, YT) > 0.0

{ Determine which side has most room for the key, and derive the
{ the location of the key. First, see if the key is vertical.
```

```
      IF MAX(XL,XR) >= MAX(YB,YT)
         WIDTH = MIN(0.4*MAX(XL,XR), 0.25*(DX2-DX1))
         HEIGHT = MIN(6.0*WIDTH, 0.7*(DY2-DY1))
         IF XL > XR
            XK1 = DX1 - 1.5 * WIDTH
            XK2 = DX1 - 0.5 * WIDTH
         ELSE
            XK1 = DX2 + 0.5 * WIDTH
            XK2 = DX2 + 1.5 * WIDTH
         ENDIF
         YK1 = 0.5 * (DY2 + DY1 - HEIGHT)
         YK2 = 0.5 * (DY2 + DY1 + HEIGHT)
      ELSE

{ Deal with horizontal key.
         WIDTH = MIN(0.4 * MAX(YB,YT), 0.25 * (DY2-DY1))
         HEIGHT = MIN(6.0 * WIDTH, 0.7 * (DX2-DX1))
         IF YB > YT
            YK1 = DY1 - 1.5 * WIDTH
            YK2 = DY1 - 0.5 * WIDTH
         ELSE
            YK1 = DY2 + 0.5 * WIDTH
            YK2 = DY2 + 1.5 * WIDTH
         ENDIF
         XK1 = 0.5 * (DX2 + DX1 - HEIGHT)
         XK2 = 0.5 * (DX2 + DX1 + HEIGHT)
      ENDIF

{ Draw the key to fit within the current picture annotating with
{ the scaling used in DISPLAY.
      LUTVIEW LOW=(LOW) HIGH=(HIGH) LBOUND=[(XK1&','&YK1)] ~
        UBOUND=[(XK2&','&YK2)] MODE=XY
   ENDIF
END PROC
```

Chapter 10
HDS/NDF — The Data System

10.1 HDS — Hierarchical data system

HDS — the Hierarchical Data System — is one of the most powerful features of ADAM. It is implemented as a set of subroutines which are of much more interest to the programmer than the user of the programs. Nevertheless, as a user it is necessary for you to know something of the system in order to make the best use of your data. HDS is about storing astronomical data in a compact, flexible and efficient way. It recognises that observations are often complex — possibly consisting of a data array (in 1, 2, 3, or even more dimensions), together with variable amounts of ancillary data — calibrations, errors, telescope and instrument information, observing conditions, and so on. The way HDS handles this complexity bears some similarities to the way VMS handles directories and files.

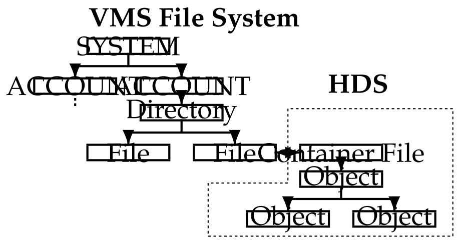

Figure 10.1: The relationship between VMS and HDS.

10.1.1 Data objects

HDS files are known as *container files* and by default have the extension '.SDF'. They contain *data objects* which will be referred to simply as *objects* when the context makes clear what sort of object it is. An object is an entity which contains data or other objects. This is the basis of the hierarchical nature of HDS and is analogous to the VMS concepts of *file* and *directory* — a directory can contain files and directories which can themselves contain files and directories and so on (Figure 10.1). An object possesses the following attributes:

- Name
- Type
- Shape
- State
- Group
- Value

HDS allows great freedom in specifying names and types, but standards have been laid down (see Section 10.2) to encourage portability of data and applications.

Name:

An object is identified by its *name*. This must be unique within its own container object. This is in contrast to VMS where different files in the same directory may be distinguished by their version numbers. A name is written as a character string containing any printing characters. Spaces, tabs and so on are ignored and alphabetic characters are capitalised. There are no special rules governing the first character (i.e. it can be numeric).

When referring to components of objects, the following syntax is used:

```
    A.B.C...
```

where C is a component of B, and B is a component of A, which is the top-level object in the container file. Some specific examples of names are given at the end of this section.

Type:

The *type* of an object falls into one of two *classes*:

- Primitive
- Structure

Structure objects contain other objects called *components*. Primitive objects contain only numeric, character, or logical values. Objects in the different classes will be referred to simply as *structures* and *primitives*, while the more general term *object* will refer to either a *structure* or a *primitive*. Structures are analogous to VMS directories — they can contain a part of the hierarchy below them. Primitives are analogous to VMS files — they are at the bottom of any branch of the structure.

The primitive types defined in HDS are shown in Table 10.1.

HDS Type	*VAX Fortran Type*	*Length in Bits*
_INTEGER	INTEGER	32
_REAL	REAL	32
_DOUBLE	DOUBLE PRECISION	64
_LOGICAL	LOGICAL	32
_CHAR[*N]	CHARACTER*N	8*N
_UBYTE	BYTE	8
_BYTE	BYTE	8
_UWORD	INTEGER*2	16
_WORD	INTEGER*2	16

Table 10.1: The HDS primitive data types.

The first five of these types are referred to as *standard data types*. The _UBYTE type provides a value range of 0 to 255; the _UWORD type provides a value range of 0 to 65535. The others are as for Fortran 77. Examples of structure types are IMAGE, SPECTRUM, INSTR_RESP *etc.* Their names don't begin with an underscore, so the system and the programmer can easily distinguish between primitives and structures. A *type* is written as a character string with the same rules as for *name*, except that an asterisk can only appear if the first character is an underscore (i.e. it is a primitive), and also a type can be blank.

Shape:

Every object has a *shape* or dimensionality. This is described by an integer (the number of dimensions) and an integer array (the size of each dimension). A *scalar*, for example a single number, has by convention a dimensionality of zero, i.e. number of dimensions is 0. A *vector* has a dimensionality of 1, i.e. number of dimensions is 1 and the first element of the dimension array contains the size of the vector. An *array* refers to an object with 2 or more dimensions; currently a maximum of 7 dimensions are allowed. Objects may be referred to as *scalar primitives* or *vector structures* and so on.

State:

The *state* of an object specifies whether or not its value is defined. In routines it is represented as a LOGICAL variable where .TRUE. means defined and .FALSE. means undefined.

Group:

In order to access an object, it is first necessary to obtain a *locator*, a sort of pointer which can then be used to address the object. When the program no longer needs to access the object, the locator should be *annulled*. A locator is analogous to a Fortran logical unit number (but is actually a character variable, not an integer). Any number of locators can be active simultaneously. The *group* attribute is used to form an association between locators so that they can be annulled together. A group is written as a character string whose rules of formation are the same as for *name*.

Value:

When an object is first created it contains no value, somewhat like an empty file. It must be given a value in a separate operation. A value can be a scalar, vector, or an array. The scalar or the elements of the vector or array must all be of the same type and can be primitives or structures. The rules for handling character values are the same as for Fortran 77, i.e. character values are padded with blanks or truncated from the right depending on the relative length of the program value and the object.

Illustration:

To fix ideas, look at the example of an NDF data structure in Figure 8.2. The following notation is used to describe each object:

```
                      NAME(dimensions)  TYPE  VALUE
```

where '`(dimensions)`' only appears when describing vectors or arrays. Each level down the hierarchy is indented.

Suppose an object with this structure were stored in a (container) file called EXAMPLE.SDF, then we can refer to components of this object by names such as:

```
      EXAMPLE.DATA_ARRAY            an array of type _REAL
      EXAMPLE.QUALITY.BADBITS       an unsigned scalar of type _BYTE
      EXAMPLE.MORE.FIGARO.TIME      a scalar of type _REAL
```

and so on.

10.2 NDF — Extensible n-dimensional data format

A major preoccupation of Starlink since its inception has been to design a data storage format which is both standard and yet which can accommodate most of the things which one might wish to store. (This is a weak point with most software environments in astronomical use at present.) One of the practical problems with unfettered HDS is that it is *too* flexible. The solution adopted, NDF (Extensible *N*-dimensional-Data Format), provides a more limited set of designs, but still implemented using HDS. It is described in awesome detail in SGP/38.

In essence, NDF defines a set of standard data objects. Not all of them must be present in an NDF object, but no others will be processed. Non-standard items are handled in a standard way by using self-contained *extensions*. There are defined locations for items such as the main data array, axes, title, units *etc.* The only mandatory item is the main data array; all other items are optional!

All this means that the *user* can be certain that no properly written application will mess up his data, and there is a very good chance that all the useful information will be properly used. (For the *programmer*, the huge advantage of this system is that he doesn't need to know the details of the format at all! A comprehensive set of routines is available to access the standard components of an NDF. These are described in Section 21.2.1 and (more fully) in SUN/33.)

10.2.1 The structure of an NDF object

ADAM_EXAMPLES:EXAMPLE.SDF is file containing an NDF object which contains all the standard NDF components and also has a Figaro extension. Such a file is often referred to as an 'NDF file', or even as just an 'NDF'. The structure of the file, as revealed by:

```
ICL > TRACE ADAM_EXAMPLES:EXAMPLE
```

is shown in Figure 8.2.

```
EXAMPLE  <NDF>

   DATA_ARRAY(856)  <_REAL>       *,0.2284551,-2.040089,
                                  ... 820.8976,570.0729,*,449.574
   TITLE           <_CHAR*30>     'HR6259 - AAT fibre data'
   LABEL           <_CHAR*20>     'Flux'
   UNITS           <_CHAR*20>     'Counts/s'
   QUALITY         <QUALITY>      {structure}
      BADBITS         <_UBYTE>       1
      QUALITY(856)    <_UBYTE>       1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,
                                     ... 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0

   VARIANCE(856)   <_REAL>        2.1,0.1713413,1.5301,34.38378,42.35531,
                                  ... 615.6732,427.5547,353.9127,337.1805
   AXIS(1)         <AXIS>         {structure}

   Contents of AXIS(1)
      DATA_ARRAY(856)  <_REAL>       3847.142,3847.672,3848.201,3848.731,
                                     ... 4298.309,4298.838,4299.368,4299.897
      LABEL           <_CHAR*20>     'Wavelength'
      UNITS           <_CHAR*20>     'Angstroms'

   HISTORY         <HISTORY>      {structure}
```

```
         CREATED        <_CHAR*30>      '1990-DEC-12 08:21:02.324'
         CURRENT_RECORD  <_INTEGER>     3
         RECORDS(10)    <HIST_REC>      {array of structures}

         Contents of RECORDS(1)
            TEXT           <_CHAR*40>      'Extracted spectrum from fibre data.'
            DATE           <_CHAR*25>      '1990-DEC-19 08:43:03.08'
            COMMAND        <_CHAR*30>      'FIGARO V2.4 FINDSP command'


      MORE           <EXT>           {structure}
         FIGARO         <EXT>           {structure}
            TIME           <_REAL>         1275
            SECZ           <_REAL>         2.13
```

Figure 8.2: An example of the internal structure of an NDF file.

Of course, this is only an example format. There are various ways of representing some of the components. These *variants* are described in SGP/38.

The components of an NDF are described below. The names (in bold type) are significant as they are used by the NDF access routines to identify the components.

DATA_ARRAY — the main data array is the only component which *must* be present in an NDF. In the case of EXAMPLE.SDF, this component is a 1-d real array with 856 elements.

TITLE — the character string `'HR6259 - AAT fibre data'` describes the contents of the NDF. The TITLE might be used as the title of a graph *etc.*

LABEL — the character string `'Flux'` describes the quantity represented in the NDF's main data array. The LABEL is intended for use on the axes of graphs *etc.*

UNITS — this character string describes the physical units of the quantity stored in the main data array, in this case, `'Counts/s'`.

QUALITY — this component is used to indicate the quality of each element in the main data array. The quality structure contains a quality array and a BADBITS value, both of which *must* be of type _UBYTE. The quality array has the same shape and size as the main data array, and is used in conjunction with the BADBITS value to decide the quality of a pixel in the main data array. In the example the BADBITS component has value 1. QUALITY normally works by taking the *bit-wise AND* of BADBITS with each element of the QUALITY array. Thus, an odd value in the QUALITY array indicates a bad value, while an even value identifies a good pixel.

VARIANCE — the variance array is the same shape and size as the main data array and contains the errors associated with the individual data values. These are stored as *variance* estimates for each pixel.

AXIS — this structure may contain axis information for any dimension of the NDF's main array. In this case, the main data array is only 1-d, therefore only the AXIS(1) structure is present. This structure contains the actual axis data array, and also label and units information.

HISTORY — this component provides a record of the processing history of the NDF. Only the first of three records is shown in the example. This indicates that the spectrum was extracted from fibre data using the Figaro FINDSP command on 19th December 1990. (Support for the history component is not yet provided by the NDF access routines.)

EXTENSIONs — the purpose of extensions is to store non-standard items. `EXAMPLE.SDF` began life as an old-style (DST) Figaro file[1] which contained values for the airmass and

[1]The file was converted to an NDF using the CONVERT command DST2NDF.

exposure time associated with the observations. These are stored in the Figaro extension, and the intention is that the Figaro applications which use these values will know where to find them.

Part IV

FOR PROGRAMMERS

Chapter 11
A Guided Tour

This chapter shows you how to write programs for the ADAM environment, and how to compile, link, and test them. It starts off with an ultra-simple 'Hello, world' program, and then takes you through other examples, explaining what is going on. Later sections consider error handling, and how to combine several programs together into a composite program called a *Monolith*.

A comprehensive description of how to write ADAM programs is given in SUN/101.

11.1 A 'Hello, world' program

Let's write, compile, link, and run an ADAM program which writes 'Hello world' on your terminal. Here's the program:

```
      SUBROUTINE HELLO(STATUS)
      INTEGER STATUS
      CALL MSG_OUT('MESS', 'Hello, world', STATUS)
      END
```

Store this in a file called `HELLO.FOR`.

You also need an *interface file* which, in its simplest form, is:

```
interface HELLO
endinterface
```

Store this is a file called `HELLO.IFL`.

Now, prepare the environment for ADAM program development:

```
$ ADAMSTART
$ ADAMDEV
```

and compile the program:

```
$ FORTRAN HELLO
```

link it:

```
$ ALINK HELLO
```

and run it:

```
$ RUN HELLO
Hello, world
$
```

You can also run the program from the ICL command language:

```
$ ICL
...
ICL> define hello hello
ICL> hello
Loading HELLO into xxxxHELLO
Hello, world
ICL> exit
$
```

You have now prepared, compiled, linked and run your first ADAM program.

11.2 Source code

The source code of every ADAM program which is meant to last should be written in the style recommended by Starlink; in particular, it should contain a section giving information about the function of the program and the meaning of the parameters. This style is encapsulated in a set of *Prologues*, described in section 12.2. Once you start serious ADAM programming, you should base your code on these prologues. Their purpose is to ensure that your code is adequately documented and, in particular, can be supported easily by someone other than yourself.

The simple example programs in this chapter use a much simpler format than that recommended for normal ADAM programs as their purpose is pedagogic. They are usually presented first without comments in order to make their structure as clear as possible. Then, the code is explained line by line.

ADAM programs are written as Fortran *subroutines* with one argument — a status value. They should obey ADAM conventions; in particular, all communication with the user must be done via the parameter, message, or error systems, and the data system should be used to manipulate data. You must not use READ or WRITE statements to communicate with the user's terminal directly. This is because a direct WRITE would bypass ADAM's system of output control (which may be using a screen management system like SMS), and a direct READ would bypass the parameter system for obtaining parameter values. Furthermore, the sub-process in which the program is run may not be connected to a terminal.

An example of the source code for a simple program to read a real value from a terminal and write it out is:

```
      SUBROUTINE TESTR(STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER STATUS
      REAL XVALUE
*.........................................................................
      CALL PAR_GETOR('X', XVALUE, STATUS)
      IF (STATUS.EQ.SAI__OK) THEN
         CALL MSG_SETR('X', XVALUE)
         CALL MSG_OUT('MESS','Value from TESTR program is ^X',STATUS)
      END IF
      END
```

In these teaching programs I use a horizontal line of dots to separate the declarations from the executable statements. This is not standard ADAM practice, but I think it makes the structure of the code easier to comprehend.

You can tell this is an ADAM program because it calls routines which implement the ADAM parameter and message systems. Let's go through it line by line and see what is going on.

```
          SUBROUTINE TESTR(STATUS)
```

As already mentioned, the program is written as a Fortran subroutine with a single parameter called STATUS; the name of the subroutine is TESTR.

```
          IMPLICIT NONE
```

This makes the compiler tell us if we are using a variable which hasn't been declared. It is a useful way of trapping spelling mistakes and declaration omissions. It is good programming practice to declare explicitly every variable used.

```
          INCLUDE 'SAE_PAR'
```

Before you can compile an ADAM program, you need to execute a command called ADAMDEV. Amongst other things, this defines a logical name called SAE_PAR to be the name of a file which contains the definitions of Fortran global constants which are needed in ADAM programs. You should always include this INCLUDE statement in your programs[1].

```
          INTEGER STATUS
          REAL XVALUE
```

These statements declare the types of the two variables, STATUS and XVALUE, used in the program.

```
          CALL PAR_GET0R('X', XVALUE, STATUS)
```

This is the first executable statement, and is a call to one of the routines which implement the parameter system — you can tell this because its name starts with the characters 'PAR_'. The rest of the name tells you its function: 'GET' means 'get the value of a parameter'; '0' means 'the parameter has 0 dimensions' (i.e. it is a scalar); 'R' means 'present the parameter value to the program as a REAL number'. There is an extensive repertoire of similar routines with names like PAR_GET1I and PAR_PUTNR whose meanings can be broken down in a similar way. In fact you can specify the dimensionality of the parameter value to be:

0 — meaning 'scalar'

1 — meaning 'vector'

N — meaning 'n-dimensional'

V — meaning 'map object as if it were a vector'

and you can specify the type to be:

D — meaning DOUBLE PRECISION

R — meaning REAL

I — meaning INTEGER

L — meaning LOGICAL

C — meaning CHARACTER[*n]

[1]These last two statements (IMPLICIT and INCLUDE) can be replaced by the single statement INCLUDE 'SAI_PAR'. However, it is clearer if they are kept separate when explaining what is going on.

The PAR routines are described in APN/6 and summarised in section 21.1.

But what do the subroutine arguments stand for?

Well, the first argument 'X' is a CHARACTER expression specifying the *name* of the parameter whose value is being obtained. The next argument 'XVALUE' is the name of the REAL scalar variable which is to hold the *value* obtained for the parameter. The final argument 'STATUS' is an INTEGER variable which will hold the value of the Status returned by the routine.

To sum up: this statement obtains the value of parameter X from the ADAM parameter system, stores it as a REAL number in scalar variable XVALUE, and stores the returned status value in variable STATUS.

```
      IF (STATUS.EQ.SAI__OK) THEN
```

Now we test the status value returned by PAR_GET0R by comparing it with the constant SAI__OK. This is one of those constants defined as a result of that 'INCLUDE 'SAE_PAR'' statement we came across earlier. SAI__OK means that 'no error has been detected', so if this test is satisfied we can execute the next two statements which cause the value read in to be displayed on the user's terminal:

```
      CALL MSG_SETR('X', XVALUE)
      CALL MSG_OUT('MESS', 'Value from TESTR program is ^X', STATUS)
```

These two routines belong to the ADAM message system, which is the preferred way of displaying messages on the user's terminal. Once again, the first four characters 'MSG_' show that they are message system routines, and the rest of the name indicates their function. The values of variables are passed to the message system by means of 'tokens', so the first thing to do is to set the value of a token. This is done by MSG_SETR which encodes the value of the REAL variable XVALUE and associates it with the token specified by the CHARACTER expression 'X'. There is a different routine for each type of variable (MSG_SETL, MSG_SETI etc.).

N.B. The argument 'X' used in PAR_GET0R and MSG_SETR give names to different things. In the case of PAR_GET0R, it is the name of a program parameter. In the case of MSG_SETR, it is the name of a message system token. The program doesn't get confused because each routine interprets 'X' in its own way.

The MSG_OUT routine constructs the message and writes it on the user's terminal. `'MESS'` is a CHARACTER expression specifying the name of the message in the message system; `'Value from TESTR program is ^X'` is the message itself, where ^ indicates the token which will be replaced by the value set by MSG_SETR; and STATUS holds the status value returned by the routine. The MSG routines are described in Chapter 16.

Finally:

```
      END IF
      END
```

terminate the IF statement and indicate the end of the program source code.

This program should be stored in a file called TESTR.FOR, ready for compiling. However, before the program can be run successfully, we need to prepare an *interface file*.

11.3 Interface file

The interface file contains information on a program's parameters and messages, and shields the program from details of the run-time environment which may not be known when the program is written — in particular, a program can ask for a parameter value without knowing how it will be obtained. The interface file should be called *program*.IFL where *program* is the name of the program. Notice that the interface file can be changed without having to change and recompile the program with which it is associated — useful flexibility. An example interface file for the TESTR program above is:

```
interface TESTR
  parameter X
    type     '_REAL'
    position 1
    prompt   'x value'
    ppath    'current,default'
    default  1.5
    vpath    'prompt'
  endparameter
  message MESS
     text 'TESTR prints ^X'
  endmessage
endinterface
```

The format, content, and meaning of interface files are described in detail in Chapter 14. However, it should be clear that they begin and end with '`interface`' and '`endinterface`' statements respectively. They give information about parameters (`parameter ... endparameter`) and messages (`message ... endmessage`).

The example parameter specification (beginning '`parameter X`') contains details about the TESTR program's single parameter (named X) and how it should be treated. It does this by giving values for a number of *fields* in the format (*field-name value*). The field specifications shown above have the following effect:

```
type     '_REAL'
```

The data type of the parameter value is _REAL. (This is one of the primitive object types in the HDS data system.)

```
position 1
```

A value for X may be given in the *first* parameter position on the command line.

```
prompt   'x value'
```

The *prompt-string* that is to be presented to the user when the system asks for a value is 'x value'.

```
ppath    'current,default'
```

'ppath' is short for 'prompt-value-resolution-path', and the purpose of this field is to specify where the suggested value offered to the user in the prompt is to come from. In this case, the current (last used) value will be used or, if there is no current value, the default value specified in the interface file will be used.

```
default  1.5
```

The default value is 1.5. Notice that, because of the 'ppath' specification above, this value will only appear in the prompt as the suggested value if there is no current value.

```
vpath    'prompt'
```

'vpath' is short for 'value-resolution-path', and the purpose of this field is to specify the search path the system is to follow when it is trying to obtain a value for a parameter. If a value is specified on the command line, the problem is solved and 'vpath' is not considered. However, if no value is specified, 'vpath' gives the system an ordered list of alternative sources to try. In the example above, only one source is specified: prompt the user to specify a value. Be careful to distinguish between the meanings of 'vpath' and 'ppath'; 'ppath' is concerned with the *prompt* — hence the 'p', while 'vpath' is concerned with the *value* — hence the 'v'.

The prompt will be of the form:

> *keyword* - *prompt-string* /*suggested-value*/ >

In the example above, the *keyword* is taken as the 'X' in the 'parameter' statement, the *prompt-string* is specified by the 'prompt' statement, and the *suggested-value* is determined by the 'ppath' statement. Thus, if there is no current value, the prompt for parameter X would be:

```
X - x value /1.5/ >
```

By changing the specification of 'vpath' and/or 'ppath' in the interface file, the program can be made to accept a value obtained from a variety of sources.

The message specification:

```
message MESS
   text 'TESTR prints ^X'
endmessage
```

tells ADAM that when the program outputs the message with the name 'MESS', the message '`TESTR prints x`' (where `x` is the value associated with the message token `^X` by the program) is to be displayed in preference to the message given in the program (i.e. in the source code). If the message specification is omitted from the interface file, the text given in the source code would be displayed (i.e. '`Value from TESTR program is x`').

Now that we have the source code stored in file TESTR.FOR and the interface module stored in file TESTR.IFL, we are ready to compile, link, and test our program TESTR.

11.4 Compiling and linking

The ADAM linking process links the subroutine with any other user-supplied or ADAM routines which are called, together with a fixed part which handles program startup, shutdown etc.

Before compiling and linking an ADAM program, the following commands must first be executed to set up the required logical names and symbols:

```
$ ADAMSTART
  ...
$ ADAMDEV
+ logged in for ADAM program development
$
```

(You may already have executed the '`ADAMSTART`' command, in which case you don't need to execute it again.) You compile the program in the normal way:

```
    $ FORTRAN TESTR
```

producing the object code in file TESTR.OBJ. This can now be linked with the ADAM environment by:

```
    $ ALINK TESTR
```

to produce the executable file TESTR.EXE. The ALINK command is defined during the execution of ADAMDEV.

11.5 Testing

The TESTR program can now be tested using the command language ICL. First, start up ICL as shown in section 5.2, then a possible test session is as follows:

```
    ICL> define demo testr
    ICL> demo 5.1
    Loading TESTR into xxxxTEST
    TESTR prints 5.1
    ICL>
```

Let us consider this test session one command at a time. First, it is necessary to define a command to run the program — this is done by the command:

```
    ICL> define demo testr
```

Here, 'demo' is the name of the command being defined, and 'testr' is the program to be run when the command is issued. For this to work, program TESTR has to be stored in one of the directories in the ADAM_EXE searchlist (such as your default directory). If it is stored somewhere else, a directory specification must be specified in front of 'testr' in the 'define' command. The next command:

```
    ICL> demo 5.1
    Loading TESTR into xxxxTEST
```

causes our program TESTR to be loaded into subprocess xxxxTEST (as shown in the second line) and executed. As the parameter value (5.1) is provided on the command line, the interface file 'vpath' specification for parameter X is not used, i.e. the user is not prompted for a value. On execution, the program displays the message:

```
    TESTR prints 5.1
```

This was obtained from the specification for message MESS in the interface file, and not from the message stored in the MSG_OUT call in the program. Now try:

```
    ICL> demo
    X - x value /5.1/ > 4
    TESTR prints 4
    ICL>
```

Notice that TESTR does not require re-loading (there is no loading message). As a value for X was not specified on the command line, the system displays a prompt in response to the '`vpath 'prompt'`' field specification in the interface file. The *suggested-value* (5.1) is the current (last used) value rather than the default (1.5) value because of the order specified in the '`ppath 'current,default'`' field specification in the interface file. The new value (4) is accepted and output in the message on the following line.

Testing from DCL:

It is possible to run the program directly from DCL by:

```
$ run testr
X - x value /1.5/ > 4
TESTR prints 4
$
```

Programs for different data types:

Program TESTR can be modified to read a parameter of a different type. Thus we could create a program TESTI to read integer values:

```
      SUBROUTINE TESTI(STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER STATUS
      INTEGER XVALUE
*.........................................................................
      CALL PAR_GET0I('X', XVALUE, STATUS)
      IF (STATUS.EQ.SAI__OK) THEN
         CALL MSG_SETI('X', XVALUE)
         CALL MSG_OUT('MESS', 'Value from TESTI program is ^X', STATUS)
      END IF
      END
```

with an interface file like:

```
interface TESTI
  parameter X
    type     '_INTEGER'
    position 1
    prompt   'x value'
    ppath    'current,default'
    default  1
    vpath    'prompt'
  endparameter
  message MESS
     text 'TESTI prints ^X'
  endmessage
endinterface
```

Similarly, we could write programs TESTL and TESTC to read and write logical and character type parameters. These can be used to explore the response of the parameter system and ICL command processor to different types of input value. These TESTx programs can also be combined into a single program called a *monolith* — this is demonstrated later in Section 11.8.

11.6 Error handling

All but the simplest ADAM programs should be structured as a top level routine which calls one or more subroutines which, in turn, may call further routines. For example, consider this abbreviated sketch of the subroutine structure of a program to add two images together:

```
SUBROUTINE ADD(STATUS)
...
CALL GETINP('INPIC1', LOCI1, STATUS)
CALL GETINP('INPIC2', LOCI2, STATUS)
CALL CREOUT('OUTPIC', 'OTITLE', NDIMS1, DIMS1, LOCO, STATUS)
CALL CMP_MAPV(LOCO,'DATA_ARRAY','_REAL','WRITE',PNTRO,DIMTOT,STATUS)
CALL ADDARR(DIMTOT, %VAL(PNTRI1), %VAL(PNTRI2), %VAL(PNTRO), STATUS)
...
END

SUBROUTINE GETINP(PARNAM, LOCAT, STATUS)
...
CALL DAT_ASSOC(PARNAM, 'READ', LOCAT, STATUS)
...
END

SUBROUTINE CREOUT(PARNAM, TLENAM, NDIM, DIMS, LOCAT, STATUS)
...
CALL DAT_NEW(LOCAT, 'DATA_ARRAY', '_REAL', NDIM, DIMS, STATUS)
...
END

SUBROUTINE ADDARR(DIMS, INARR1, INARR2, OUTARR, STATUS)
...
END
```

Being an ADAM program, it calls routines (like CMP_MAPV, DAT_ASSOC, DAT_NEW) in the ADAM libraries. Such routines have an integer parameter called STATUS and if they fail for some reason, STATUS will be set to an error code indicating the nature of the error, otherwise its value will remain unchanged. Our private routines (such as GETINP, CREOUT, ADDARR) should also have a STATUS parameter which should be set to an error code if an error is detected.

The treatment of the STATUS parameter is governed by the *ADAM Error Strategy*. This is:

- Check the value of STATUS immediately on entry. If its value is not SAI__OK, return immediately without doing any more processing.
- Leave STATUS unchanged if the routine completes successfully.
- Set STATUS to an appropriate error number if an error is detected.

The application of this strategy can be illustrated once again by the following program:

```
SUBROUTINE ADD(STATUS)
...
IF (STATUS.NE.SAI__OK) RETURN
  <top level control>
END

SUBROUTINE GETINP(PARNAM, LOCAT, STATUS)
...
IF (STATUS.EQ.SAI__OK) THEN
  <get a locator to an IMAGE type structure for data input>
END IF
END

SUBROUTINE CREOUT(PARNAM, TLENAM, NDIM, DIMS, LOCAT, STATUS)
...
```

```
      IF (STATUS.EQ.SAI__OK) THEN
        <create and return a locator to an IMAGE type structure>
      END IF
      END

      SUBROUTINE ADDARR(DIMS, INARR1, INARR2, OUTARR, STATUS)
      ...
      IF (STATUS.EQ.SAI__OK) THEN
        <add two arrays>
      END IF
      END
```

For each subroutine, the IF statement is the first *executable* statement. This error strategy means that it is usually not necessary to check STATUS after each routine is called. A series of routines can be called with STATUS being passed from one to the next. If an error occurs in one of them, the subsequent routines will do nothing and the final STATUS will indicate the error code from the routine that failed. If this value is then returned by the main routine to the fixed part, an error message will result. Thus, the error will be correctly processed with no special code being added to check for errors.

A program will always be executed by the ADAM system with an initial STATUS value of SAI__OK. This is because during the linking process it is linked with a 'fixed part' which calls it as a subroutine after having initialised STATUS to this value. Thus, it would appear to be unnecessary to test STATUS on entry to our program. However, sometime in the future we may want to call our program as a subroutine in another program and it might be called after a previous routine has set STATUS to an error code. Thus, in general, we cannot be sure what the value of STATUS on entry to our program will be, and we should *always* adopt the Starlink error strategy, even for our top-level routines.

This error strategy is illustrated by our next example program which calculates the square of the value of the input parameter 'VALUE', and writes it out as part of a message:

```
      SUBROUTINE SQUARE(STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER STATUS
      REAL R, RR
*.....................................................................
      IF (STATUS.NE.SAI__OK) RETURN
      CALL PAR_GETOR('VALUE', R, STATUS)
      IF (STATUS.EQ.SAI__OK) THEN
         RR = R*R
         CALL MSG_SETR('RVAL', R)
         CALL MSG_SETR('RSQUARED', RR)
         CALL MSG_OUT(' ', 'The Square of ^RVAL is ^RSQUARED', STATUS)
      END IF
      END
```

Here, STATUS is used in two tests. Firstly:

```
      IF (STATUS.NE.SAI__OK) RETURN
```

to implement the ADAM Error Strategy, and secondly:

```
      IF (STATUS.EQ.SAI__OK) THEN
         ...
      END IF
```

to ensure that if the STATUS returned from PAR_GET0R is bad, the rest of the routine is not executed with an undefined value of R. Actually, it is not necessary to include MSG_OUT in the IF block as this routine would return immediately if STATUS was bad.

An example interface file for this program is:

```
interface SQUARE
  parameter VALUE
    type     '_REAL'
    position 1
    prompt   'Number to be squared'
    ppath    'current'
    vpath    'prompt'
  endparameter
endinterface
```

Notice that the first parameter of the MSG_OUT routine is specified as a single blank character. This means that the message being output does not have a name and there is no specification for it in the interface file. This is the simplest method of using the message system, but it means that we cannot alter the message by specifying it in the interface file.

To test this example, enter the source code and interface file into files SQUARE.FOR and SQUARE.IFL, respectively; compile and link as for TEST, then use the following commands within ICL:

```
ICL> define square square
ICL> square 12
Loading SQUARE into xxxSQUARE
The Square of 12 is 144
ICL> square (sqrt(3))
The Square of 1.732051 is 3
ICL> square
VALUE - Number to be squared /1.732051/ > 7
The Square of 7 is 49
ICL>
```

More sophisticated error handling can be provided by using routines in the ERR library. These facilities are described in Chapter 16.

11.7 Returning parameter values

We have seen how routines like PAR_GET0R get parameter values from the environment. It is also possible for programs to return values *to* the environment. The following modified fragment of program SQUARE does not output its result on the terminal (by using MSG_OUT), but returns it to the parameter VALUE using a call to the routine PAR_PUT0R, which is analogous to PAR_GET0R. Replace the second IF statement by:

```
IF (STATUS.EQ.SAI__OK) THEN
   RR = R*R
   CALL PAR_PUT0R('VALUE', RR, STATUS)
END IF
```

We could run the modified program from ICL as follows:

```
ICL> x=5
ICL> square (x)
ICL> =x
        25
ICL>
```

In order for the program to return a value to ICL, we must use a variable for the parameter and place it on the command line:

```
ICL> square (x)
```

The variable name must be placed in parentheses; the name of a temporary data object holding the value of the variable is used as the parameter by ICL.

A modification of this scheme is needed with character variables to allow for the case where the value of the character variable is itself a device, file or object name. In such cases, the supplied name cannot be replaced by some other name so, to indicate that they may not be replaced, name values in variables must be preceded by '@'. For example, in

```
ICL> x='devdataset'
ICL> trace (x)
```

the character string 'devdataset' would be stored in a temporary data object and the effect of 'TRACE (X)' would be to trace the temporary object and not devdataset. However, in

```
ICL> x='@devdataset'
ICL> trace (x)
```

devdataset itself will be traced.

11.8 Monoliths

We have shown how to write ADAM programs called TESTR, TESTI, TESTL, and TESTC to read and display parameter values of four different types. It would be convenient to combine these four similar programs in a single program which would recognise the individual commands which call them. This can be done by combining them into a *monolith*. KAPPA is an example of a monolith in which a lot of small programs have been combined together. The advantage is that once the monolith has been loaded, all the component programs can be used without any further loading operations being necessary, and it only occupies one place in the task cache.

Let us produce a monolith called TEST which will contain the four TESTx programs mentioned above. The first thing to do is to create an object library to hold the object modules for the TESTx programs:

```
$ LIB/CREATE REDUCE
```

This will create an object library called REDUCE.OLB. Assuming the four TESTx programs have been compiled, we can store their object modules in this library by:

```
$ LIB REDUCE TESTC,TESTI,TESTL,TESTR
```

Now, write a program to call the TESTx programs in response to appropriate commands:

```
      SUBROUTINE TEST(NAME, STATUS)
      CHARACTER*(*) NAME
      INTEGER STATUS
*...........................................................................
      IF (STATUS.NE.SAI__OK) RETURN
      IF (NAME.EQ.'TESTC') THEN
        CALL TESTC(STATUS)
      ELSE IF (NAME.EQ.'TESTI') THEN
        CALL TESTI(STATUS)
      ELSE IF (NAME.EQ.'TESTL') THEN
        CALL TESTL(STATUS)
      ELSE IF (NAME.EQ.'TESTR') THEN
        CALL TESTR(STATUS)
      END IF
      END
```

Store this in file TEST.FOR, then compile it:

```
$ FOR TEST
```

and link it with the routines it calls by using the MLINK command:

```
$ MLINK TEST,REDUCE/LIB
```

(we must have executed the ADAMDEV statement in order to define the symbol MLINK). We now have the monolith stored in file TEST.EXE, but we also need an interface file stored in file TEST.IFL. This is simply a concatenation of the interface files for the TESTx programs enclosed in 'monolith' and 'endmonolith' statements:

```
monolith TEST
interface TESTC
  parameter X
    type     '_CHAR'
    position 1
    prompt   'x value'
    ppath    'current,default'
    default  'Default'
    vpath    'prompt'
  endparameter
  message MESS
     text 'TESTC prints ^X'
  endmessage
endinterface
interface TESTI
  parameter X
    type     '_INTEGER'
    position 1
    prompt   'x value'
    ppath    'current,default'
    default  1
    vpath    'prompt'
  endparameter
  message MESS
     text 'TESTI prints ^X'
  endmessage
endinterface
interface TESTL
```

```
      parameter X
        type     '_LOGICAL'
        position 1
        prompt   'x value'
        ppath    'current,default'
        default  TRUE
        vpath    'prompt'
      endparameter
      message MESS
         text 'TESTL prints ^X'
      endmessage
    endinterface
    interface TESTR
      parameter X
        type     '_REAL'
        position 1
        prompt   'x value'
        ppath    'current,default'
        default  1.5
        vpath    'prompt'
      endparameter
      message MESS
         text 'TESTR prints ^X'
      endmessage
    endinterface
    endmonolith
```

The last thing to do before TEST is ready for use is to store definitions for the commands TESTC, TESTI, TESTL, TESTR in an ICL command file TEST.ICL:

```
define testc test
define testi test
define testl test
define testr test
```

Now, we are ready to use our monolith TEST. Start up ICL in the usual way, then execute the commands which define the commands to run the programs by loading the command file TEST.ICL:

```
ICL> load test
```

Now, if any of the defined commands 'testc', 'testi', 'testl', 'testr' is entered, the monolith 'test' will be loaded:

```
ICL> testc
Loading TEST into xxxxTEST
X - x value /'Default'/ > yogi bear
TESTC prints yogi bear
ICL> testr
X - x value /1.5/ 7.8
TESTR prints 7.8
ICL>
```

Notice that the monolith TEST is only loaded once and that the command 'testr' is available for immediate use. The extra delay in loading the larger monolith and in defining the set of commands is made up for later by the faster response to subsequent commands.

Chapter 12
Programming Standards, Conventions and Tools

The most striking features of the *Starlink Software Collection* are its size (in March 1990 it contained about 1 million lines of code, half a million comment lines, and another half million blank lines) and the number of programmers who have contributed to it. These are both its strength and its weakness. In particular, it presents a growing maintenance problem. The only hope of keeping it under control is by 'keeping up standards'. This includes both the sense of 'better' rather than 'worse', and also 'this is what BSI, ANSI, ISO . . . have to say on the subject'.

While nobody wants to stifle a programmer's creativity, you have to recognise that software constructed using the guidelines discussed here is likely to be a great deal easier to maintain than that written as the spirit moves you. Software which is very personal to its creator is likely to suffer one of two fates: it will not be used by anyone else, or the original programmer will have the task of maintaining it for life.

Software portability has always been regarded by the Starlink Project as important, but has, in fact, been of marginal interest to the majority of users. Now, however, the spread of Starlink to Unix-based systems means that portability is a very real problem for a growing number of users.

The aim of these standards is to achieve software which, while providing the best possible applications, is also portable, maintainable by anyone, and integrates with the other software in the Collection. The software tools provided should make it easier to produce compliant code than not!

12.1 Language standards

12.1.1 Fortran

SGP/16 offers advice on how to write application programs for Starlink, embodied in a set of general rules. As a programmer you should bear in mind the needs of the person who will eventually be responsible for maintaining your code. You are strongly encouraged to read SGP/16 if you have not already done so.

ADAM has a number of special requirements which may mean that one of the general rules has to be reinterpreted — in some cases strengthened, in others relaxed. There are, in addition, several new rules which do not have to be obeyed when writing non-ADAM applications. Here is a summary of the extra rules which apply to ADAM applications, as compared with ordinary free-standing programs:

- Initialise variables.

  It is **essential** that variables are initialised. Even the VAX's initialisation to 0 cannot be relied on as the task may or may not be reloaded between invocations. DATA statements must only be used to initialise data which will not change.

- Use the ADAM standard prologues

  ADAM standard prologues differ in some respects from the Starlink standard, allowing less freedom but giving more opportunity for the automatic production of documentation and help files. Standard prologues exist for subroutines and interface files.

- Output messages via the MSG routines.

  Message output must be done using the ADAM Message system (MSG) subroutines.

- Don't use $, %, ^ in messages.

  The non-Fortran 77 characters $, % and ^ are used as escape characters in the ADAM message system.

- Report errors via the ERR routines.

  Error reporting must be done using the ADAM Error system (ERR) subroutines, and the ADAM Error Strategy should be employed.

- Set STATUS on failure.

  All programs which fail must return to the environment with an error status value set. This enables the environment to detect the failure so that users can write procedures which take appropriate action.

- When setting STATUS, generate a message.

  If a subroutine is entered with STATUS=SAI__OK but, during execution, sets the STATUS (other than by calling another ADAM routine), an appropriate error message must be generated using ERR_REP.

- Some routines have $>$ 6 character names.

  Some of the ADAM environment package subroutines have names and prefixes greater than 6 characters. Where it is necessary to call these, the general rules must be relaxed.

- Get parameters with the PAR routines etc.

  All program parameters must be obtained using the Parameter system PAR or pkg_ASSOC subroutines.

- Use symbols when testing for bad pixels.

  A REAL or DOUBLE PRECISION variable may be equated to its corresponding bad-pixel value, though explicit bad-pixel values, e.g. -32767, are banned. The parameters VAL__BADx, where x corresponds to the data type, must be used.

- Avoid Fortran input/output.

  Use the environment facility packages MAG, FIO etc. wherever possible. If it is necessary to use Fortran I/O, obtain and release logical unit numbers using FIO_GUNIT and FIO_PUNIT.

- Use symbolic names.

  Status values and package constants are given symbolic names such as PAR__NULL by INCLUDE files for each package. These symbolic names should be used on every occasion that the constant is required.

- RETURN is permissible when testing status.

  The RETURN statement is allowed in the form:

  ```
  IF (STATUS.NE.SAI__OK) RETURN
  ```

  as the first executable statement in a subroutine. This avoids an extra, unhelpful IF clause and indentation. Alternatively, use a GO TO n, where line n is a CONTINUE statement immediately preceding the END statement.

- In generic routines use only the standard tokens.

  The preprocessor for generic routines supports special tokens used by the ASTERIX package (SUN/98), as well as ones for general use. Use only the standard tokens.

- PAR__ABORT status (!!) must abort the application.

  An application must terminate if the *abort* response (!!) is made when a parameter has been requested. Note that this rule does not mean that the application has to test for the abort status after every parameter is obtained; the inherited status will look after that. What matters is the appearance to the user of the application, who should:

  – not be re-prompted for the parameter,
  – not be prompted for further parameters, and
  – not receive additional error messages merely because the status was not OK.

  An abort does not absolve the programmer from ensuring that the application closes down in an orderly fashion.

12.1.2 C

In spite of the first rule (use Fortran) in the Starlink Application Programming Standard, SGP/16, there is a Starlink C Programming Standard — SGP/4. Its general style and philosophy are very similar to the Fortran standard, though obviously many of the rules are only directly relevant to one of the languages.

The suggestions which it contains are made with maintainability, portability and efficiency in mind. Several of them are made because certain code constructs will have different effects in different C implementations. In some cases the ambiguity is resolved by the ANSI standard. However, it may be some time before all C implementations meet the standard, and in any case it is preferable to avoid code which suggests more than one possibility to the human reader.

12.1.3 Fortran/C interface

There are always problems when writing programs in a mixture of Fortran and C. Each vendor offers his own solution, but the problem for Starlink is to try to provide a portable solution. SGP/5 describes two packages to help with this problem:

F77 — is a set of C macros to handle the Fortran/C subroutine linkage.

CNF — is a set of C functions to handle the difference between Fortran and C character strings.

This software is currently available on VAX/VMS systems, Sun SPARC systems, and DEC systems running Ultrix/RISC (typically, DECstations).

12.1.4 Posix interface

You may need to use C simply to call one of its run-time library routines, to allocate memory for example. A Fortran library called PSX has been provided to enable you to avoid doing this (see SUN/121). In many programs, it will remove the need to write any C code at all.

PSX allows you to use the functions provided by the Posix and X/Open libraries.

12.2 Prologue standards

Starlink has defined a number of standard prologues as a starting point for programming. They are important because they provide a minimal level of source-code documentation, and they can also be processed into on-line and off-line documentation by software tools. These prologues are normally accessed through STARLSE (see Section 12.3). However, they are also available as files which you can edit. These can be found in `STARLSE_DIR` in files with the following names:

ATASK.PRO — A-task template.

SUB.PRO — subroutine template.

FUNC.PRO — function template.

BLOCK.PRO — block data routine template.

MON.PRO — monolith template.

IFL.PRO — interface file template.

As an example of a complete template, here is BLOCK.PRO (blank lines have been removed to save space):

```
      BLOCK DATA {routine_name}
*+
*  Name:
*     {routine_name}
*  Purpose:
*     {routine_purpose}
*  Language:
*     {routine_language}
*  Type of Module:
*     BLOCK DATA
*  Description:
*     {routine_description}
*  Notes:
*     {routine_notes}...
*  Side Effects:
*     {routine_side_effects}...
*  Implementation Deficiencies:
*     {routine_deficiencies}...
*  {machine}-specific features used:
*     {routine_machine_specifics}...
*  {DIY_prologue_heading}:
*     {DIY_prologue_text}
*  References:
*     {routine_references}...
*  Keywords:
*     {routine_keywords}...
*  Copyright:
*     {routine_copyright}
*  Authors:
*     {author_identifier}: {authors_name} ({affiliation})
*     {enter_new_authors_here}
*  History:
*     {date} ({author_identifier}):
*        Original version.
*     {enter_changes_here}
*  Bugs:
*     {note_any_bugs_here}
*-
*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing
*  Global Constants:
      [standard_SAE_constants]
      INCLUDE '{global_constants_file}' ! [global_constants_description]
*  Global Variables:
      INCLUDE '{global_variables_file}' ! [global_variables_description]
*        {global_name}[dimensions] = {data_type} ({global_access_mode})
*          [global_variable_purpose]
*  Local Constants:
      {data_type} {constant_name} ! [constant_description]
      PARAMETER ( {constant_name} = {cons} )
*  Local Variables:
      {data_type} {name}[dimensions] ! [local_variable_description]
*  Global Data:
      DATA {data_elm} / {data_values}... /
*.
      END
```

This contains *everything* you might need, though many of the items will not always be needed.

12.3 Software tools

Software tools are programs which help you write programs. Of the ones described below, STARLSE and SST are specific to ADAM programs. The others can be used for any type of program.

12.3.1 FORCHECK [SUN/73]

A Fortran verifier and programming aid installed on the Starlink central facilities computer (STADAT). It checks code for conformance to the ANSI standard X3.9–1978. However, it can also deal with non-standard code and by default accepts VAX Fortran. It:

- Checks inter-module consistency by checking that the number, type, and size of elements in both sub-program argument lists and COMMON blocks are the same throughout a program.
- Identifies recursive calls and misuse of arguments.
- Identifies 'clutter': the unused variables, COMMON blocks, INCLUDE files, and code fragments which accumulate in old programs, and which make maintenance such a time-consuming and costly task.
- Composes cross-reference charts for constants, variables, COMMONs, INCLUDE files, sub-programs, and I/O.

Automatically composed documentation of this type is an invaluable addition to system documentation. As often as not, it is the only reliable source of information about old programs.

12.3.2 GENERIC [SUN/7]

Preprocesses a *generic* Fortran subroutine — one written so as to apply to several different data types — into one routine per data type, and concatenates them into a file. This can then be compiled to produce an object module.

12.3.3 LIBMAINT [SUN/99]

Simplifies the maintenance of software held as modules in a source/object library pair, and of modules in a Help library. New libraries can be created and modules inserted, extracted, replaced, examined, and printed with simple commands. It ensures that corresponding modules within a source/object library pair do not get out of step with one another, and it can optimise the disk space used by libraries.

12.3.4 LIBX [SUN/8]

Contains two tools for use with libraries. The first outputs a list of all modules in a library, and is useful for building more elaborate utility procedures. The second extracts preamble comments from all the modules in a Fortran source library.

To some extent they duplicate facilities in LIBMAINT. The latter is a very powerful system but, unavoidably, is vulnerable to changes in the formats of the reports which the DEC Librarian utility produces. The LIBX facilities, though limited in what they do, use only published interfaces and should survive new VMS releases; they are also fast.

12.3.5 SPAG [SUN/63]

SPAG stands for 'Spaghetti Unscrambler'. It re-orders blocks of Fortran statements in such a way that the structure of the code is improved, while remaining logically equivalent to the original program. The result improves the readability and maintainability of badly-written Fortran programs. On Starlink, it may be used to convert unstructured Fortran 77 into the structured and indented VAX Fortran required by the Starlink Programming Standard. It can also be used to update Fortran 66 code. It is marketed by Polyhedron Software and is available only on the Starlink central facilities machine, STADAT.

12.3.6 SST [SUN/110]

The Simple Software Tools package was described in Chapter 4. However, its importance can best be appreciated when programming. In particular, three of the tools — PROHLP, PROLAT, and PROPAK — process prologues in useful ways:

PROHLP — converts prologues into on-line Help text. Suppose you have source code in file PROG.FOR and you are building up the Help library MYLIB.HLB, then:

```
ICL> PROHLP PROG.FOR PROG.HLP
ICL> $ LIBRARY/HELP MYLIB.HLB PROG.HLP
ICL> $ DELETE PROG.HLP;*
```

will add more material.

PROLAT — converts prologues to LaTeX form, which can be used as either a stand-alone document, or incorporated into another document.

PROPAK — converts prologues from the code of a subroutine package into a form that STARLSE can use. STARLSE can be 'taught' about a subroutine library, so that simply typing the name of a routine (or just the first few characters) followed by ctrl/E will expand it into either a list of routines (if the string entered is not unique), or into a call to that routine ready for its arguments to be filled in. This removes a common source of error.

12.3.7 STARLSE [SUN/105]

This is a 'Starlink Sensitive' editor based on the VAX Language Sensitive Editor LSE. It helps you write portable Fortran 77 software in a standard Starlink style.

If you normally use a standard screen editor (like EDT), you will probably find that using an LSE-based editor for the first time will slow you down considerably. All those new keys to remember! However, once you get used to it, you will start to realise how much time you used to spend doing simple things like moving the cursor, formatting prologues, indenting lines of code and going to fetch essential documentation — all things which STARLSE can do far more efficiently.

One of the strengths of LSE-based editing is that it allows you to pick and choose — to use the features that you personally find time-saving, while still being able to type directly over anything which you find too fussy. In this respect, STARLSE can be regarded as a sort of interactive 'Manual of Style' which provides guidelines on layout and programming standards when you need reminding of them, but lets you work unhindered once you know what you are doing.

Perhaps the most important component of STARLSE is a version of the Fortran 77 programming language called STARLINK_FORTRAN, which defines the style of programming that the editor supports. This language is used by default for files of type .FOR and .GEN.

The language is based on the Starlink Application Programming Standard and contains only a small number of approved extensions to Fortran 77. It is therefore much simpler and easier to use than the VAX Fortran language which comes with 'native' LSE (in fact, nearly 80% of VAX Fortran consists of extensions to the Fortran 77 standard!). In STARLINK_FORTRAN, the number of available options and ambiguous abbreviations is greatly reduced, and most of the common language constructs can be produced simply by typing a short token, like 'DO' or 'IF', followed by ctrl-E.

The most important features of the language are:

- Templates and Prologues.
- Subroutine definitions.
- On-line Help.
- Alias definitions.
- ADAM programming constructs.
- Symbolic constants, error codes, and Include files.
- Enumerated type codes.
- Tokens and Menus.

For further information, refer to SUN/105 and the DEC LSE User Guide and Reference Manual.

12.3.8 TOOLPACK [SUN/75]

Toolpack/1 (release 2) is a suite of software tools designed to support the Fortran programmer. In this context, a 'software tool' is a utility program to assist in the various phases of constructing, analysing, testing, adapting, or maintaining a body of Fortran software. Typically, the input to such a tool is your Fortran source code. The tool processes this and produces output that may have one or both of the following forms:

- A report that gives an analysis of the input program, e.g. a summary of the types of statements used; this type of tool is called a static analyser.
- A modified version of the input program; in this case, the tool is called a transformer. An example is a formatter which improves the appearance of the code.

In some cases the input may be test data, documentation, or a report generated by a previously applied tool. Tools that assist directly in preparing documents are usually called documentation generation aids. These and other tools serving utility functions all have an important role to play and so, even if they do not process a program directly, they are still regarded as programming aids.

Further examples of the software tools provided include:

- A text editor with Fortran 77 oriented features.
- A transforming tool that standardises the declarative part of a Fortran program.
- An instrumenter that modifies the program by inserting monitoring and other control statements. The instrumented program is then compiled and executed, and data is gathered that is used to generate reports. Execution of an instrumented program is an example of dynamic analysis.

If you want to know more about TOOLPACK, read the 'Introductory Guide' available from your Site Manager.

Chapter 13
The ADAM Libraries

ADAM provides subroutine libraries which make various facilities available to application programmers. Each library has a *Facility Name* which is a mnemonic used to identify it. For example, the Facility Name for one of the libraries in the Starlink Data System is 'DAT'. The ways in which these Facility Names impact upon the programmer are:

Routine Names —
Each routine in a library has a name of the form:

```
fac_name
```

where 'fac' is the Facility Name, and 'name' is the specific routine name. By using this convention, name clashes between different libraries and applications can be avoided. It also becomes possible to identify the ADAM calls in a piece of code.

Error Symbols —
The error codes which can be returned via the STATUS argument are given symbolic names using Fortran PARAMETER statements. These names are made available to a program by including a file with logical name 'fac_ERR', as in:

```
INCLUDE 'DAT_ERR'
```

which includes the symbolic names of error codes returned by the Data System (DAT). This file need only be included by routines which need to identify the precise nature of an error. The symbolic names have the form:

```
fac__error
```

where 'error' is a mnemonic used to identify the error (note the use of two '_' (underline) characters). The actual values of the error codes originate from the VMS MESSAGE utility, and so form a coherent set with the status returns from VMS libraries. A set of error codes which may be used by writers of non-ADAM facility libraries is provided in the include file USER_ERR.

Parametric Constants —
These are symbolic names given to the constants associated with a library. The most commonly needed constants are defined in a file with logical name 'SAE_PAR'. In the situation where a program requires the parametric constants for other facilities, these must be included by:

```
INCLUDE 'fac_PAR'
```

The form of these symbolic constants is identical to that used for error codes, namely:

```
fac__const
```

This chapter provides an overview of the most important ADAM libraries. They are grouped into systems. Each system is described in more detail in a subsequent chapter except the database system, which is mainly used in the SCAR applications package, and the utilities, which are described in Starlink User Notes.

13.1 Parameter system

This provides the mechanism for controlling the action of ADAM programs:

PAR [AED/15, APN/6]
This provides high-level access to the parameter system. It makes it easy to input a small number of data values from the terminal and to specify the data objects, devices and so on which the program requires.

13.2 Data system

This provides the dominant mechanism by which programs store and manipulate data. The data system as a whole is sometimes referred to colloquially as HDS, and sometimes as NDF. The following facilities are available, listed using a top-down approach:

NDF [SUN/33, SGP/38]
This stands for Extensible *N*-dimensional Data Format, as described in Chapter 10. It is the standard Starlink format for storing data arrays. As such, it is expected to form the basis of most Starlink 'image' processing applications, where 'images' include arrays which have more or less than two dimensions.

ARY [SUN/11, SGP/38]
This is a set of routines for accessing Starlink ARRAY data structures built using HDS. The most likely reason for using these routines directly is to access ARRAY structures stored in NDF extensions.

REF [SUN/31]
These routines allow you to store references to HDS data objects in special HDS reference objects, and allow locators to reference objects to be obtained. Their main uses are to maintain a catalogue of HDS objects and to avoid duplicating a large dataset.

HDS [SUN/92]
The Hierarchical Data System is a flexible system for storing and retrieving data, and is of great importance to the whole of the Starlink Project. It is the basic data system in ADAM and is also used in many other software items. It is used very extensively in the implementation of ADAM itself and is also used by application programmers. The routines themselves are mainly concerned with the highest level of the data system, including the *container file* which is the interface between the data system and the host computer's file system.

CMP [SUN/92]
These routines simplify the coding needed to access *structure* components within HDS.

DAT [SUN/92, APN/7]
These routines access and manipulate data objects, which may be *primitive* (e.g. arrays of numbers), or *structured* (i.e. collections of other objects).

13.3 Message and Error systems

These related systems provide the *only* mechanism by which text should be sent to the command device (user terminal or batch log file). Output will normally go to the command device via a user-interface. The facilities involved are:

MSG [SUN/104]

Reports non-error information to the user.

ERR [SUN/104]

Reports error messages to the user and environment.

EMS [SSN/4]

Constructs and stores error messages, but doesn't communicate them to the user. It is, therefore, of no interest to the application programmer, but of considerable interest to the programmer who is building new *systems* software. It is included here for completeness.

13.4 Graphics system

This consists of several facilities offering different levels of control, but all (except IDI) are based on the ISO standard Graphics Kernel System (GKS).

NCAR/SNX [SUN/88, SUN/90, MUD/59]

An extensive suite of high level graphics utilities originating from the National Center for Atmospheric Research in Boulder, Colorado. SNX contains some Starlink-produced extensions to NCAR.

PGPLOT [SUN/15, MUD/61]

A high level package for plotting x-y plots, functions, histograms, bar charts, contour maps and grey-scale images. The version in use on Starlink uses GKS for its low-level graphics input/output.

NAG graphics [SUN/29]

The NAG Graphics Library (formerly the *Graphical Supplement*) was originally seen as a way of plotting data associated with the numerical routines which make up the main NAG library. However, it can be used quite independently of the main library for plotting graphs, functions, and contours, though most people now prefer to use NCAR or PGPLOT.

SGS [SUN/85]

A 'simplified' graphics facility which provides most of the commonly needed basic graphics functions in a convenient form. The primary simplification over GKS is that, while several workstations can be open simultaneously, only one can be active at a given time. It has been designed so that calls to its routines can be freely interspersed with those of GKS. Specialised GKS functions are not reproduced in SGS.

GKS [SUN/83, MUD/27]

The basic set of 'implementation' routines. It is a fairly powerful low-level graphics facility, and is *the* international 2-dimensional graphics standard.

IDI [SUN/65]

A standard for displaying astronomical data on an image display. It complements, rather than replaces, GKS, and should be used where intimate control of the image display device is required, and for functions which are outside the scope of GKS.

AGI [SUN/48]

A database system which stores information about plots on a graphics device. This enables a program to relate to plots produced by other programs.

GNS [SUN/57]

Almost every Starlink graphics program will need the name of a GKS or IDI device. GNS — the Graphics Name Service — allows the programmer to choose a device from a reasonably 'friendly' series of names, which is then translated into the GKS 'workstation type'. Unless you are opening GKS workstations directly or making specialised enquiries about devices, you are most unlikely to need to call GNS yourself.

13.5 Input/output systems

In general, data storage should be handled through the data system. However, there may be a need to handle files directly. This is provided by the following closely related facilities:

FIO [SUN/143]
Handles sequential, formatted files for the production of reports, e.g. for subsequent listing. The form of carriage control may be specified when the file is created.

RIO [SUN/143]
Handles unformatted, direct-access files.

There may also be a need to access 'foreign' magnetic tapes:

MAG [APN/1]
Handles tape positioning, and reading or writing data blocks and tape marks. It also includes facilities for keeping track of tape positions so that programs can provide a friendly user interface.

13.6 Database system

These are provided mainly for access to astronomical catalogues:

CHI [SUN/119]
This is gradually replacing ADC as the preferred method of accessing relational databases, such as the usual astronomical catalogues or large tables of measurements of sets of objects.

ADC
Provides access to the relational database files handled by the SCAR programs. It is implemented as part of the SCAR software item.

13.7 Utilities

These differ from the libraries mentioned above in that they are not associated with parameters. They are libraries which enable ADAM programs to do various tasks in a consistent way:

CHR SUN/40 Manipulation of character strings, encoding and decoding numeric fields etc.

CNF SGP/5 Interchange of character strings between Fortran and C software modules.

PRIMDAT [SUN/39]
Arithmetic, mathematical operations, type conversion, and inter-comparison of any of the primitive data types supported by HDS.

PSX [SUN/121]
: An interface to the Posix portable operating system.

SLALIB [SUN/67]
: Routines mainly concerned with astronomical position and time.

TRANSFORM [SUN/61]
: A standard, flexible method for manipulating coordinate transformations, and for transferring information about them between programs.

Chapter 14
The Parameter System

The Parameter system is a central feature of the ADAM system architecture. It is a unifying concept which binds the various ADAM 'facilities' together and permeates many aspects of ADAM. Both users and programmers need to understand it clearly if they are to use ADAM properly.

A list of the routines which implement the system is given in Section 21.1.

14.1 Parameters and objects

Programs are usually written to be as general as possible in order to maximize their usefulness. Thus, you would not write a program to display a specific image but would make it accept *any* image stored in one or more specified formats. Because of this, at run time the program needs to know the name of the image to be displayed. The 'name of the image' is a *parameter* of the program. More complicated programs may have many parameters controlling their actions. Thus, your display program may need to know the format of the input image, the device on which it is to be displayed, the colour mappings to be used in the display, types of axes, scaling methods, and so on. All these things are parameters. When an ADAM program is run, the task of the ADAM parameter system is to associate the program's parameters with objects in the outside world.

The word 'object' is rather overworked in ADAM, so you have to be aware of the context in which it is used. 'Object' is often used to mean a 'data object', *i.e.* an HDS structure accessed through the data system. When talking about parameters, the word 'object' has a wider meaning; in fact it can mean absolutely anything outside itself that a program wishes to refer to by name — a data object, a hardware device, a message, a graphics database, a file *etc.*

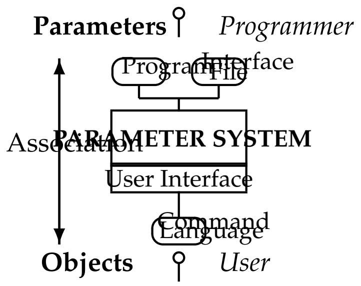

Figure 14.1: The Parameter System — Communication between User and Programmer.

Figure 14.1 is a schematic illustration of the function of the ADAM Parameter System. Its main role is to provide a means of communication between the user and the programmer, and in particular to associate objects chosen by the user with a program's parameters chosen by the programmer. The programmer communicates by using ADAM routines in his program and parameter specifications in his interface file. The user communicates by using a command language such as ICL.

You must be careful to distinguish between a 'parameter name' (used to identify which parameter is being talked about), and a 'parameter value' (the value a given parameter has at a particular moment).

The word ‘parameter’ is often used loosely in both senses. However, the precise meaning is normally obvious from the context.

14.2 Associating objects with parameters

The programmer has two ways to associate an object with a program parameter:

- Directly, using ‘PAR’ routines.
- Indirectly, using ‘ASSOC’ routines.

Direct association:

The direct method uses the set of PAR_GET and PAR_PUT routines introduced in Chapter 11. It is called ‘direct’ because a call to a single routine causes the parameter value to be read into a program variable, or written from a program variable. Thus, in the program TESTR illustrated in Chapter 11 the call:

```
    CALL PAR_GETOR('X', XVALUE, STATUS)
```

causes the parameter system to obtain a real scalar value for parameter ‘X’ and store it in program variable ‘XVALUE’. Similarly, a call such as:

```
    CALL PAR_PUTOR('Y', YVALUE, STATUS)
```

causes the parameter system to write the value stored in program variable ‘YVALUE’ into the data object associated with parameter ‘Y’.

The direct method only works for parameters whose values are scalars, vectors, or n-dimensional arrays of one of the five standard HDS primitive data types:

- _INTEGER
- _REAL
- _DOUBLE
- _LOGICAL
- _CHAR

Values of other types (*e.g.* graphics plots, non-standard data types) must be associated with parameters by the indirect method.

Indirect association:

The various ADAM subroutine libraries enable you to access and manipulate objects of various kinds. The indirect method of accessing parameters first ‘associates’ an object of an appropriate type with a parameter by calling a special ‘ASSOC’ routine provided for that library. This routine provides the program with an ‘identifier’. From then on, this identifier is used to specify the object, rather than the parameter name.

The usual form[1] of the ASSOC routine is:

[1]The ASSOC routines for the FIO and RIO libraries are slightly unusual in that they need two extra parameters: the format, FM, and the record size, RS. These are placed after MODE.

```
    CALL fac_ASSOC(PAR, MODE, ID, STATUS)
```

where:

fac is the facility name of the library.

PAR is the parameter name.

MODE is the access mode permitted to the object.

ID is the identifier used to identify the object in other routines in the library.

STATUS is the status return.

Table 14.1 shows the names of the ASSOC routines for the ADAM subroutine libraries, together with the name of the object identifier 'ID' used in the documentation, and the type of object addressed.

Subroutine library	*ASSOC routine*	*Object identifier*	*Object type*
NDF,DAT,CMP,HDS,REF,TRN	DAT_ASSOC	LOC	Data object
GKS	GKS_ASSOC	WKID	GKS workstation
SGS	SGS_ASSOC	ZONE	SGS zone
IDI	IDI_ASSOC	DISPID	Display device
AGI	AGI_ASSOC	PICID	Picture in database
FIO,RIO	FIO_ASSOC	FD	File
MAG	MAG_ASSOC	TD	Tape drive
ADC	ADC_ASSOC	CD	Catalogue
NCAR,SNX,PGPLOT,GNS,NAG	not specified		
PAR,MSG,ERR,EMS	not needed		

Table 14.1: ASSOC routines for ADAM subroutine libraries.

Once an object has been associated with a parameter, its identifier is used to access the object. Suppose, for example, that parameter 'IMAGE' is a data object which contains a pixel array stored in a component named 'DATA_ARRAY'. This array can be accessed by the code:

```
    CALL DAT_ASSOC('IMAGE', 'READ', LOC, STATUS)
    CALL DAT_FIND(LOC, 'DATA_ARRAY', LOCA, STATUS)
    CALL DAT_GETNR(LOCA, NDIM, DIMX, VALUE, DIM, STATUS)
```

in which the first call associates an HDS object with the parameter 'IMAGE' and makes the locator 'LOC' point to it. The second call finds the component 'DATA_ARRAY' in this image and makes another locator 'LOCA' point to it. The third call stores the value of the array in variable 'VALUE'.

Most facilities also have ANNUL routines which are used to stop the identifier pointing to the object, thus:

```
    CALL DAT_ANNUL(LOC, STATUS)
```

will cause LOC to stop pointing to the object associated with parameter 'IMAGE'. The object will still be associated with parameter 'IMAGE', but you won't be able to access it. It is considered good practice to annul identifiers when you have finished using their objects. However, the system tidies up after you when your program finishes so you needn't worry about it.

The ASSOC and CANCL routines (described later) are often referred to as 'Parameter Routines' because they are added to a 'stand-alone' subroutine library in order to integrate it with the ADAM parameter system. The equivalent 'stand-alone' routines are OPEN and CLOSE. In ADAM programs the OPEN routine is usually replaced by the ASSOC routine. For example, when the SGS graphics library is used outside ADAM, the first call to one of its routines would be:

```
    CALL SGS_OPEN(WKSTN, ZONE, STATUS)
```

which would cause the graphics workstation specified by WKSTN to be opened and an SGS zone specified by ZONE to be made available for plots. However, an ADAM program which used SGS would not use this call, but would replace it with:

```
    CALL SGS_ASSOC(PAR, MODE, ZONE, STATUS)
```

Here, the workstation is specified through the parameter system, so PAR replaces WKSTN in the calling sequence. ZONE and STATUS have the same function as before, but a new variable, MODE, is introduced to provide the program with the ability either to clear or to retain the display screen contents.

Parameter values:

Your program relies on the parameter system to provide a value for a parameter. This process depends on a 'search path' defined in the interface file, as will be explained later. However, your program can influence the value provided by suggesting a 'dynamic default' to the parameter system. This is done by a set of 'PAR_DEF' commands. For example:

```
    CALL PAR_DEFOR('X', 3.4, STATUS)
```

will set a dynamic default of 3.4 for parameter 'X'.

The set of PAR_DEF routines is analogous to the PAR_GET and PAR_PUT sets, except that there is no 'V' form, *i.e.* you cannot set a dynamic default for an array mapped as a vector; you have to use the explicit array form.

14.3 Interface files

An interface file for a single program has the following form:

```
      interface PROGNAME
         Parameter specification
         Parameter specification
         ...
         Message specification
         Message specification
         ...
      endinterface
```

The body of the file comes between the 'interface' and 'endinterface' statements, and consists of a sequence of 'Parameter specifications' and 'Message specifications'. Each interface file has a unique name (PROGNAME) which should be the same as the name of the program with which it is associated. Comments may be introduced by the '#' character — any characters following '#' on a line will be ignored.

Minimal interface files:

The example interface file shown in Section 11.3 was quite large considering the simplicity of its associated program. Is all this complexity really essential? The answer is 'No' — the example given was made complex in order to show off the facilities of ADAM, and not because of any intrinsic complexity in using ADAM. Suppose you write the following abbreviated ADAM program to read in a real number and write it out on your terminal:

```
      SUBROUTINE READ(STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER STATUS
      REAL XVALUE
*.....................................................................
      CALL PAR_GETOR('X', XVALUE, STATUS)
      CALL MSG_SETR('X', XVALUE)
      CALL MSG_OUT('MESS', 'Value read for X is ^X', STATUS)
      END
```

Now, suppose you don't prepare an interface file for this program at all and try to run it using ICL in the usual way. This is what would happen:

```
ICL> define read read
ICL> read
Loading READ into xxxxREAD
Failed to read interface module
...
```

When I tried it, the system hung and I had to use cntrl/Y to escape. So, you can't get away with not storing an interface file because the system just won't work without one. However, suppose you store one with no information on the program parameter X, *i.e.* an interface file containing the lines:

```
interface READ
endinterface
```

If you try to run the program again it still won't work, but you will get a different error message:

```
ADAMERR   %PARSECON, requested parameter name not defined
```

However, at least the system won't hang. Now let's go one step further and put an empty specification for parameter X in the interface file:

```
interface READ
  parameter X
  endparameter
endinterface
```

This time the program will work, even though we haven't stored a specification for the message parameter 'MESS':

```
ICL> read
X   > 1.2
Value read for X is 1.2
ICL>
```

So, the shortest interface file you can get away with is one which contains empty specifications for all the program parameters except message parameters (such as MESS) which do not need to be specified at all. Thus, the simplest way of testing a program you have compiled and linked is to prepare an interface file containing empty specifications for its non-message parameters, then use ICL to define a command to run it, and then type in that command. You will be prompted for a value for each of the program's parameters. If you try to specify values on the command line by position, they will be ignored and you will be prompted anyway. However, you can specify them on the command line by keyword, using the parameter name as the keyword.

You can see now that the function of the interface file is to provide you with the ability to modify or enhance the user interface if required. The facilities available from the interface file are to:

- Control the data type of a value associated with a parameter.
- Control the access to a parameter value.
- Control the format of the command line.
- Control the appearance of prompt and help lines.
- Control the search for a parameter value.
- Control the contents of messages.

These facilities are considered in more detail in the next sections where the fieldnames which are used to specify these controls are described.

Search path:

An interface file can be accessed through a search path. If a *job* logical name search path, ADAM_IFL, is defined, it will be used to find the interface file. If ADAM_IFL is not defined, or if an interface file is not found using it, an attempt will be made to find one in the directory in which the program executable image was found.

Compilation:

An interface file can exist in two forms. The first form is an ordinary text file which can be edited easily. This is used when a program is being developed. The file should have the same name as the program, but with a file extension of '.IFL'. When the user interface for a program has become relatively stable, the interface file should be compiled and stored as an '.IFC' file — this is the second form. This is done as follows:

```
$ COMPIFL progname
COMPIFL: successful completion
$
```

where '`progname`' is the name of the interface file. (You must have previously entered the ADAMSTART and ADAMDEV commands.) The system can extract information from interface files stored in the second form faster than from files stored in the first form, so this improves efficiency.

14.4 Parameter specifications

The different types of fieldname provided in an interface file for controlling the user interface fall naturally into various groups which have a particular function, as shown in Table 14.2. Some of the fieldnames have slightly different functions in different contexts and so belong to more than one group. The groups and their fieldnames are described in more detail below. Each fieldname description is headed by a line with the format:

Group	*Fieldname*	*Function*
Data type & access	**TYPE**	Data type
	PTYPE	Device type
	ACCESS	Access mode
Command line	**POSITION**	Position in command line
	KEYWORD	Keyword in command line
Prompt	**KEYWORD**	Keyword in prompt
	PROMPT	Prompt text
	HELP	On-line help text
	HELPKEY	On-line help text
	PPATH	Prompt-value search path
	ASSOCIATION	Associated global parameter
	DEFAULT	Default prompt-value
Value	**VPATH**	Parameter-value search path
	ASSOCIATION	Associated global parameter
	DEFAULT	Default parameter-value
	IN	Valid value set
	RANGE	Valid value range

Table 14.2: Groups of fieldnames used in parameter specifications in interface files.

FIELDNAME `fieldvalue`

This shows how this fieldname is specified.

Data type & access group —

These fieldnames specify the data type of the parameter, and how it should be accessed:

TYPE `data-type`

This specifies the *type* of a *Normal* parameter, *i.e.* any parameter except a device type. It may used by the system to check that a supplied parameter value is of a type that the application program is expecting. It also allows automatic conversion between the primitive data types. It is important to distinguish between the type of a parameter and the type of the value to be obtained from it by the application.

The `data-type` may be:

- An HDS primitive type (*i.e.* _INTEGER, _REAL, _DOUBLE, _LOGICAL, _CHAR, _UBYTE, _BYTE, _UWORD, _WORD).
- LITERAL — formerly used to specify that the parameter is of type _CHAR, and to force the parameter system to accept unquoted character strings as character values rather than HDS object names. This is now the standard behaviour for parameters of type _CHAR, so data-type LITERAL is no longer required.
- A ‘user-defined’ type (IMAGE, SPECTRUM *etc.*).
- UNIV — used by convention when the parameter type is unimportant. Actually, this is treated the same as other ‘user-defined’ types.

- A 'facility-recommended' type (TAPE for MAG, FILENAME for FIO *etc.*). These are used by convention — any user-defined type will do.

If the TYPE field is omitted, type UNIV is assumed.

Example:

```
parameter INPIC1
  type    image
```

An acceptable value for parameter `INPIC1` is specified as being of type `image`.

PTYPE `device`

This specifies that a parameter is of the *Device* type.

Most parameters are associated with objects whose *values* belong to some data type, *e.g.* _REAL. These are known as *Normal* parameters. However, some parameters are associated with a hardware *device*, *e.g.* a magnetic tape drive or a graphics terminal. These are known as *Device* parameters. The value of a device parameter is a device name.

Example:

```
parameter PLOTTER
  ptype   device
```

The parameter `PLOTTER` is specified as being of device type.

ACCESS `access-mode`

This specifies the *mode of access* the program requires to a parameter value. Permitted values for the '`access-mode`' field are:

read — read-only access permitted.

write — write-only access permitted.

update — read and write access permitted.

Thus, if a program does not modify the value associated with a parameter, specify 'read' access. If the value is to be written, specify 'write' access. If the value is to be modified, specify 'update' access. The default value is 'update'.

Example:

```
parameter INPIC1
  access  'read'
```

The parameter `INPIC1` is specified as being restricted to '`read`' access.

Command line group —

These fieldnames specify how a parameter should be specified on a command line:

POSITION `ipos`

This specifies the *position* on a command line where a value for a parameter may appear.

The '`ipos`' field is an integer. A value of '1' specifies the first parameter position, and so on.

This fieldname must be given for every parameter capable of being specified by position on the command line; there is no default value. Each such parameter must have a different position number. Positions allocated must form a contiguous set. If this fieldname is omitted, then a keyword must be specified whenever the parameter is used on a command line.

Example:

```
parameter  INPIC1
  position 1
```

The parameter `INPIC1` is associated with the first parameter position on the command line.

KEYWORD `name`

This specifies the name by which a parameter can be identified on the command line, and by which the parameter is known to the user.

The '`name`' field specifies the keyword. This may differ from the parameter name.

This fieldname separates a program's view of its parameters from a user's view of them. It is possible for a programmer to re-write a program using completely different parameter names, but the command seen by a user can be kept the same as before by just leaving the keywords as they were. Conversely, a user's view of a program can be changed by just changing the keyword specifications in the interface file.

If a keyword is not specified for a parameter, the system will use the parameter name as the keyword in the prompt and on the command line.

Example:

```
parameter INPIC1
  keyword 'PIC1'
```

The keyword for parameter `INPIC1` is specified as being `PIC1`. Thus, on the command line we could enter:

```
ICL> ADD PIC1=ramp1
```

to specify that the input image is `ramp1`.

Prompt group —

These fieldnames determine the appearance of the prompt for a parameter value. A prompt has the following syntax:

keyword – prompt-string /suggested-value/ >

The following fieldnames are involved:

KEYWORD `name`

The keyword forms the first part of the prompt. This fieldname is described above in the command line group.

PROMPT `prompt-string`

This specifies the *prompt-string* field in the prompt for a parameter value.

This field is a character string with a maximum size of about 80 characters — it is not possible to give a single figure which is correct in all cases. The prompt-string can also be set by the program using the PAR_PROMT routine.

If this fieldname is not specified, the prompt-string field is omitted from the prompt.

Example:

```
parameter INPIC1
  prompt  'First input image'
```

The prompt-string for parameter `INPIC1` is specified as '`First input image`'.

HELP `help-text`

This gives information about the meaning of the parameter.

The '`help-text`' field is a character string containing the information. The maximum size of the field is about 132 characters — it is not possible to give a single figure which is correct in all cases.

If the user is prompted for a parameter value, a possible reply is '?'. The system responds to this by displaying the '`help-text`' on the terminal and re-prompting the user for a value.

Example:

```
parameter INPIC1
  help    'Name of IMAGE structure containing first input data array'
```

This defines the help text for parameter `INPIC1`.

HELPKEY `filename key1 key2 ...`

This specifies a help file, and module within it, at which *multi-line* help text for the parameter may be found. The text to be displayed is stored in the module defined by `key1 key2 ...` in the help file `filename`. (The keys represent the usual hierarchical keywords used to access the help system). This is replacing the earlier *one-line* help provided by the `HELP` specification.

If `filename key1 key2 ...` is replaced by `*` or `'*'`, the default values:

```
interface_name PARAMETERS parameter_name
```

will be used, where `interface_name` is the name specified in the current INTERFACE field. Use of this default implies that the library has been specified using the `HELPLIB` specification.

Example:

```
interface DISP
    helplib 'disp_dir:disp disp parameter'
    parameter IMAGE
        helpkey 'image'
```

PPATH `prompt-value-resolution-path`

This specifies the search path to be followed to obtain a value for the *suggested-value* field in the prompt. This search path is usually referred to as the 'ppath'.

If a suggested value is required, the system looks at the 'ppath' specification, picks out the first specifier, and tries to find a value from this source. If a value is not found, the next specifier is extracted and another search is made. This process continues until either a value is found, or the search path is exhausted.

The '`prompt-value-resolution-path`' field consists of a set of specifiers separated by commas. The valid specifiers are the same as for 'vpath' (see below), except that the '**prompt**' and '**noprompt**' specifiers are invalid in this context and should not be used.

If 'ppath' is not specified, a default search path of 'dynamic,default' is used.

Example:

```
parameter INPIC1
  ppath   'global,current'
```

This specifies a 'ppath' of '`global`' followed by '`current`' for parameter INPIC1.

ASSOCIATION `association-specification`

See below in the 'Value' group.

DEFAULT `default-value`

See below in the 'Value' group.

Value group —

These fieldnames determine the value given to a parameter, and check that this satisfies given criteria:

VPATH `parameter-value-resolution-path`

This specifies the search path to be followed when searching for a parameter value. This search path is usually referred to as the 'vpath'.

A value can come from many different sources, and these sources may need to be searched in different orders. The '`parameter-value-resolution-path`' field specifies these sources and the search order — this constitutes the search-path. The field consists of specifiers separated by commas. The valid specifiers are:

current — use the current value of the parameter.

default — use the default specified in the interface file.

dynamic — use the dynamic default specified by the program.

global — use the value of the associated global parameter.

noprompt — do not prompt the user for a value.

prompt — prompt the user and obtain a value.

If the value of a parameter is not specified on the command line, the system looks at the 'vpath' specification, picks out the first specifier, and tries to find a value from this source. If a value is not found, the next specifier is extracted and another search is made. This process continues until either a value is found, or the search path is exhausted. If the search path is exhausted the user is prompted for a value, unless the 'noprompt' specifier has been given, in which case the system returns a PAR__NULL status value to the program.

Example:

```
parameter INPIC1
  vpath   'dynamic,current'
```

This specifies a 'vpath' of '`dynamic`' followed by '`current`' for parameter '`INPIC1`'.

ASSOCIATION `association-specification`

This associates a program parameter with a global parameter.

If the system comes across the 'global' specifier in a vpath or ppath, this fieldname tells it which global parameter to use as a source for the value.

The '`association-specification`' field is a character string which specifies the global parameter. This string has two parts:

```
<association-operator><parameter-specification>
```

The '`association-operator`' field must one of the following:

<- (read-only).

<-> (read-write).

-> (write-only).

'write' means that if the task completes successfully, the current value of the parameter will be copied to the global parameter.

The '`parameter-specification`' field must be the name of a global parameter.

This fieldname only needs to be specified if the specifier 'global' is used in the ppath or vpath for the parameter.

Example:

```
parameter     INPIC1
  vpath       'prompt'
  ppath       'global'
  association '<->global.data_array'
```

The global parameter '`global.data_array`' will be used as the suggested-value in the prompt. The parameter value will be obtained by prompting if it is not specified on the command line.

DEFAULT `default-value`

This specifies one or more default values for a parameter.

The '`default-value`' field is a string of alphanumeric characters containing items to be used as values for the parameter. The values may be any expression acceptable to the parameter system. This fieldname must not be specified before the 'type' fieldname has been declared.

Example:

```
parameter OTITLE
  type    '_CHAR'
  vpath   'default'
  default 'KAPPA - Add'
```

The default value for parameter `OTITLE` is '`KAPPA - Add`'. The vpath specifies that this will be used as the parameter value if one is not specified on the command line.

IN `set-of-values`

This specifies a set of valid values for a parameter.

The '`set-of-values`' field consists of a list of constants separated by commas. These constants are the valid values.

Checking is carried out when a program attempts to get a value for a parameter. If the supplied value does not belong to the valid set, either the status SUBPAR__OUTRANGE is returned, or the system attempts to prompt for the value. No checking occurs when writing a value.

This fieldname should only be used for scalar parameters and must be specified after 'type'. A parameter cannot have both a '`range`' and an '`in`' field.

Example:

```
parameter BOXSIZ
  type    '_INTEGER'
  in      3, 5, 7, 9, 11, 13, 15
```

The only valid values for parameter `BOXSIZ` are 3, 5, 7, 9, 11, 13, 15.

RANGE `range-of-values`

This specifies a range of valid values for a parameter.

The '`range-of-values`' field consists of two constants separated by a comma. These constants specify the range boundaries. This fieldname has the same characteristics as the '`in`' field described above.

Example:

```
parameter SIGMA
  type    '_REAL'
  range   0.1, 5.0
```

The only valid values for parameter `SIGMA` lie in the range 0.1 to 5.0.

14.5 Message specifications

If a program uses the Message or Error systems it can specify one or more 'Message Parameters'. Text can be associated with these parameters in the interface file by using a 'Message Specification'. However, this is not essential since text specified in the program will be used by default.

The Message specification has the form:

```
message PARNAME
   fieldname fieldvalue(s)
   ...
endmessage
```

where '`PARNAME`' is the Message Parameter name which is used in the call to an ERR or MSG routine.

Currently only one fieldname is supported:

TEXT `string`

This specifies the text to be associated with the message parameter.
The '`string`' field contains the text to be used. The string may be anything acceptable to the ERR or MSG routine (*i.e.* it may include tokens *etc.*).
Example:

```
message MESS
  text  'TESTR prints ^X'
endmessage
```

This will replace the message defined in the program with the message specified by '`text`'.

14.6 Monoliths

When a set of programs are grouped into a monolith, their interface files should be concatenated into a single interface file as follows:

```
monolith MONNAME
    interface PROGNAME1
     ...
    endinterface
    interface PROGNAME2
     ...
    endinterface
    ...
 endmonolith
```

An example is given in Section 11.8.

14.7 Parameter states

A simple program will get values for its parameters, do something with them, and then finish. However, a programmer may wish to obtain a sequence of values for a single parameter, and a user may wish to indicate that no parameter value should be used. To deal with these requirements, ADAM uses the concept of *Parameter State*. A parameter can be in one of the four states shown in Table 14.3. This shows how the various states are entered.

State	*Method of entry*	*Method of getting a value*
GROUND	Program entry	Follow the vpath in the interface file
ACTIVE	GET, PUT, ASSOC routines	Use the current value
ANNULLED	Input value of '!'	Value not set; `STATUS=PAR__NULL`
CANCELLED	CANCL routines	Prompt the user for a value

Table 14.3: Parameter states and values.

In simple programs, each parameter goes from GROUND to ACTIVE state when it is given a value and stays in that state until the programs exits. The significance of the parameter state is its effect on what happens when a program asks for a value for a parameter; this is also shown in Table 14.3. If a program asks repeatedly for the value of a parameter, as in:

```
    CALL PAR_GET0R('X', XVALUE, STATUS)
    ...
    CALL PAR_GET0R('X', XVALUE, STATUS)
    ...
```

the first call will get a value for 'X' by following the vpath. However, the second and succeeding calls will obtain the same value as obtained by the first call since this is the 'current value' and the vpath is not used. In particular, the user will not be prompted for another value. However, if the programmer 'cancels' the parameter by calling PAR_CANCL before each attempt to get a value:

```
    CALL PAR_GET0R('X', XVALUE, STATUS)
    ...
    CALL PAR_CANCL('X', STATUS)
    CALL PAR_GET0R('X', XVALUE, STATUS)
    ...
    CALL PAR_CANCL('X', STATUS)
    ...
```

the user will be prompted for each call of PAR_GET0R. This technique is useful when you have finished processing one object associated with a parameter, and want to process a different object within the same program.

Each subroutine library which has been integrated with the ADAM system has a CANCL routine, identical in form to PAR_CANCL, with which to cancel the association of a parameter with an object previously made by the ASSOC routine. For example:

```
    CALL DAT_ASSOC('X', 'READ', LOC, STATUS)
      <do something with the first object associated with parameter X>
    CALL DAT_CANCL('X', STATUS)
    CALL DAT_ASSOC('X', 'READ', LOC, STATUS)
      <do something with a new object associated with parameter X>
    CALL DAT_CANCL('X', STATUS)
```

The CANCL routines are different from the ANNUL routines considered earlier. For example, in:

```
    CALL DAT_ASSOC('X', 'READ', LOC, STATUS)
      <do something with the first object associated with parameter X>
    CALL DAT_ANNUL(LOC, STATUS)
    CALL DAT_ASSOC('X', 'READ', LOC, STATUS)
      <do something with the same object associated with parameter X>
    CALL DAT_ANNUL(LOC, STATUS)
```

the second DAT_ASSOC will associate the same object with parameter X as the first one did, because DAT_ANNUL does not break the association. Notice that the first parameter of ANNUL routines is the identifier used to address the object, whereas the first parameter of CANCL routines is the parameter name.

The ANNUL routines have nothing to do with the 'ANNULLED' state of a program parameter; it's just an unfortunate clash of words.

Chapter 15 The Data System

In this chapter, a bottom-up approach is adopted — the low-level packages are described first, followed by the higher-level ones.

15.1 HDS — Hierarchical data system

15.1.1 Symbolic names and Include files

Symbolic names should be used for important constant values in HDS programs to make them clearer and to insulate them from possible future changes in their values. These symbolic names are defined by Fortran 'include' files as shown in the examples below. The following include files are available:

SAE_PAR — This file is not actually part of HDS, but it defines the global symbolic constant SAI__OK (the value of the status return indicating success) and will be required by nearly all routines which call HDS. It should normally be included as a matter of course.[1]

DAT_PAR — Defines various symbolic constants for HDS. These should be used whenever the associated value is required (typically this is when program variables are defined)

DAT_ERR — Defines symbolic names for the error status values returned by the DAT_ and HDS_ routines.

CMP_ERR — Defines symbolic names for the additional error status values returned by the CMP_ routines.

The symbolic names of these include files may be used directly on VAX/VMS systems, but on Unix systems an explicit directory specification for the file is normally also required, and the file name should appear in lower case. Thus, to include the DAT_PAR file on a VAX/VMS system, the following code would be used:

```
      INCLUDE 'DAT_PAR'
```

whereas on a Unix system, the following code is required:

```
      INCLUDE '/star/include/dat_par'
```

(/star/include is a standard directory containing include files on Starlink machines running Unix.)

If it is necessary to test for specific error conditions, the appropriate include file and symbolic names should be used in your program. Here is an example of how to use these symbols:

[1]Due to an historical anomaly on VAX/VMS systems, the file SAE_PAR also contains definitions for the DAT__ symbolic constants which should properly reside in DAT_PAR. To prevent multiple definitions occurring, DAT_PAR is therefore an empty file on VAX/VMS systems. Thus, if SAE_PAR has been included, the further inclusion of DAT_PAR is optional (but only on VAX/VMS). Its inclusion is recommended, however, because this allows the same software to be used on other systems without change.

```
      INCLUDE 'SAE_PAR'
      INCLUDE 'DAT_PAR'
      INCLUDE 'DAT_ERR'
      ...
      CHARACTER*(DAT__SZLOC) LOC1, LOC2
      CHARACTER*(DAT__SZNAM) NAME
      INTEGER STATUS
      ...
*  Find a structure component.
      CALL DAT_FIND(LOC1, NAME, LOC2, STATUS)

*  Check the status value returned.
      IF (STATUS .EQ. SAI__OK) THEN
        <normal action>
      ELSE IF (STATUS .EQ. DAT__OBJNF) THEN
        <take appropriate action for object not found>
      ELSE
        <action on other errors>
      ENDIF
```

15.1.2 Creating objects

To fix ideas, look at the example data structure in Fig 15.1. This is actually one form of NDF structure, but for the purposes of this chapter it will be treated as if it were simply an arbitrary HDS structure, i.e. we will use HDS routines rather than NDF routines to process it. The following notation is used to describe each object:

```
NAME[(dimensions)] <TYPE> [value]
```

Note that scalar objects have no dimensions and that each level down the hierarchy is indented.

```
DATASET <NDF>
   DATA_ARRAY(512,1024)     <_UBYTE>     0,0,0,1,2,3,255,3,...
   LABEL                    <_CHAR*80>   'This is the data label'
   AXIS(2) <AXIS>
      AXIS <AXIS>
         DATA_ARRAY(512)    <_REAL>      0.5,1.5,2.5,...
         LABEL              <_CHAR*30>   'Axis 1'
      AXIS <AXIS>
         DATA_ARRAY(1024)   <_REAL>      5,10,15.1,20.3,...
         LABEL              <_CHAR*10>   'Axis 2'
```

Figure 15.1: A simple NDF structure.

This example exhibits several of the most important properties of HDS data objects.

- Both structures and primitives are present in the structure.
- Scalar and non-scalar objects are present.
- You can have arrays of structures, *e.g.* the AXIS component is a vector structure (with two elements).

The following code will create the structure in Fig 15.1:

```
      INCLUDE 'SAE_PAR'
      INCLUDE 'DAT_PAR'
      CHARACTER*(DAT__SZLOC) NLOC, ALOC, CELL
      INTEGER DIMS(2), STATUS
      DATA DIMS /512, 1024/

      CALL HDS_START(STATUS)

*  Create a container file with a top level scalar object of type NDF.
      CALL HDS_NEW('dataset', 'DATASET', 'NDF', 0, 0, NLOC, STATUS)

*  Create components in the top level object.
      CALL DAT_NEW(NLOC, 'DATA_ARRAY', '_UBYTE', 2, DIMS, STATUS)
      CALL DAT_NEWC(NLOC, 'LABEL', 80, 0, 0, STATUS)
      CALL DAT_NEW(NLOC, 'AXIS', 'AXIS', 1, 2, STATUS)

*  Create components in the AXIS structure...

*  Get a locator to the AXIS component.
      CALL DAT_FIND(NLOC, 'AXIS', ALOC, STATUS)

*  Get a locator to the array cell AXIS(1).
      CALL DAT_CELL(ALOC, 1, 1, CELL, STATUS)

*  Create internal components within AXIS(1) using the CELL locator.
      CALL DAT_NEW(CELL, 'DATA_ARRAY', '_REAL', 1, DIM(1), STATUS)
      CALL DAT_NEWC(CELL, 'LABEL', 30, 0, 0, STATUS)

*  Annul the cell locator
      CALL DAT_ANNUL(CELL, STATUS)

*  Do the same for AXIS(2).
      CALL DAT_CELL(ALOC, 1, 2, CELL, STATUS)
      CALL DAT_NEW(CELL, 'DATA_ARRAY', '_REAL', 1, DIM(2), STATUS)
      CALL DAT_NEWC(CELL, 'LABEL', 10, 0, 0, STATUS)
      CALL DAT_ANNUL(CELL, STATUS)

*  Access objects which have been created.
      ...

*  Tidy up
      CALL DAT_ANNUL(ALOC, STATUS)
      CALL HDS_CLOSE(NLOC, STATUS)
      CALL HDS_STOP(STATUS)

      END
```

The following points should be borne in mind:

- The structure created is in no way static — new objects can be added or existing ones deleted at any level without disturbing what already exists. (Remember, however, that there are very strong rules about what can and cannot be put into an NDF structure. In general, the NDF routines of Section 15.3 should be used to manipulate NDFs.)
- No primitive values have been stored yet — that will be done next.

Here are some notes on particular aspects of this example:

DAT__SZLOC — This is one of the constants mentioned in Section 15.1, which is defined in the include file DAT_PAR and specifies the length in characters of all HDS locators. Similar constants, DAT__SZNAM and DAT__SZTYP, specify the maximum lengths of object names and types.

STATUS — HDS routines conform to Starlink error handling conventions and use *inherited status checking*.

HDS_NEW — A container file called 'dataset' is created (HDS provides the default file extension of '.SDF'). A scalar structure called DATASET with a type of NDF is created within this file, and a locator, NLOC, is associated with this structure. It is usually convenient, although not essential, to make the top-level object name match the container file name, as here.

DAT_NEW/DAT_NEWC — These routines create new objects within an object — they are not equivalent to HDS_NEW because they don't have any reference to the container file, only to a higher level structure. Two variants are used simply because the character string length has to be specified when creating a character object and it is normally most convenient to provide this via an additional integer argument. However, DAT_NEW may be used to create new objects of any type, including character objects. In this case the character string length would be provided via the type specification, *e.g.* '_CHAR∗15' (a character string length of one is assumed if '_CHAR' is specified alone).

DAT_FIND — After an object has been created, it is necessary to associate a locator with it before values can be inserted; this routine performs this function.

DAT_CELL — There are several routines for accessing components of objects. This one obtains a locator to a scalar object (structure or primitive) within a non-scalar object like a vector.

HDS_CLOSE — This is used to close the container file and to annul the locator passed to it.

15.1.3 Writing and reading objects

Having created a structure, the next step will usually be to put some values into it. This can be done by using the **DAT_PUT** and **DAT_PUTC** routines. For example, the main data array in the above example could be filled with values as follows:

```
      BYTE IMVALS(512, 1024)
      CHARACTER*(DAT__SZLOC) LOC

*  Put data from array IMVALS into the object DATA_ARRAY.
      CALL DAT_FIND(NLOC, 'DATA_ARRAY', LOC, STATUS)
      CALL DAT_PUT(LOC, '_UBYTE', 2, DIMS, IMVALS, STATUS)
      CALL DAT_ANNUL(LOC, STATUS)

*  Put data from character constant to the object LABEL.
      CALL DAT_FIND(NLOC, 'LABEL', LOC, STATUS)
      CALL DAT_PUTC(LOC, 0, 0, 'This is the data label', STATUS)
      CALL DAT_ANNUL(LOC, STATUS)
```

Because this sort of activity occurs quite often, *packaged* access routines have been provided (see Section 15.1.10) for the programmer.

A complementary set of routines also exists for getting data from objects back into program arrays or variables; these are the **DAT_GET** routines. Again, packaged versions exist and are often handy in reducing the number of subroutine calls required.

15.1.4 Accessing objects by mapping

Another technique for accessing the data values stored in primitive HDS objects is termed 'mapping'.[2] An important advantage is that it removes a size restriction imposed by having to declare fixed size program arrays to hold data. This simplifies software, so that a single routine can handle objects of arbitrary size without recourse to accessing subsets.

HDS provides mapped access to primitive objects via the **DAT_MAP** routines. Essentially, **DAT_MAP** will return a pointer to a region of the computer's memory in which the object's values are stored. This pointer can then be passed to another routine using the VAX Fortran '%VAL' facility.[3] An example will illustrate this:

```
      INTEGER PNTR, EL

*  Map the DATA_ARRAY component of the NDF structure as a vector of type
*  _REAL (even though the object is actually a 512 x 1024 array whose
*  elements are of type _UBYTE).
      CALL DAT_FIND(NLOC, 'DATA_ARRAY', LOC, STATUS)
      CALL DAT_MAPV(LOC, '_REAL', 'UPDATE', PNTR, EL, STATUS)

*  Pass the "array" to a subroutine.
      CALL SUB(EL, %VAL(PNTR), STATUS)

*  Unmap the object and annul the locator.
      CALL DAT_UNMAP(LOC, STATUS)
      CALL DAT_ANNUL(LOC, STATUS)

      END

*  Routine which takes the LOG of all values in a REAL array.
      SUBROUTINE SUB(N, A, STATUS)
      INCLUDE 'SAE_PAR'
      INTEGER N, STATUS
      REAL A(N)
      IF (STATUS .NE. SAI__OK) RETURN

      DO 1 I = 1, N
         A(I) = LOG(A(I))
 1    CONTINUE

      END
```

This example illustrates two features of HDS which we haven't yet mentioned:

Vectorisation — It is possible to force HDS to regard objects as vectors, irrespective of their true dimensionality. This facility was useful in the above example as it made the subroutine SUB much more general in that it could be applied to any numeric primitive object.

Automatic type conversion — The program can specify the data type it wishes to work with and the program will work even if the data are stored as a different type. HDS will (if necessary) automatically convert the data to the type required by the program.[4] This

[2]This terminology derives from the facility originally provided by VAX/VMS for *mapping* the contents of files into the computer's memory, so that they appear as if they are arrays of numbers directly accessible to a program. Although HDS still exploits this technique when appropriate, other techniques are also used internally so that HDS no longer depends on the use of file mapping (which some operating systems do not provide). The terminology remains in use, however.

[3]This VAX extension to Fortran 77 is also supported by the other implementations of Fortran for which HDS is available.

[4]This will work even if the object was originally created on a different computer which formats its numbers differently.

useful feature can greatly simplify programming — simple programs can handle all data types. Automatic conversion works on reading, writing and mapping.

Note that once a primitive has been mapped, the associated locator cannot be used to access further data until the original object is unmapped.

15.1.5 Mapping character data

Although the above example used a numeric type of '_REAL' to access the data, HDS allows any primitive type to be specified as an access type, including '_CHAR'. It gives you a choice about how to determine the length of the character strings it will map. You may either specify the length you want explicitly, *e.g:*

```
      CALL DAT_MAPV(LOC, '_CHAR*30', 'READ', PNTR, EL, STATUS)
```

(in which case HDS would map an array of character strings with each element containing 30 characters) or you may leave HDS to decide on the length required by omitting the length specification, thus:

```
      CALL DAT_MAPV(LOC, '_CHAR', 'READ', PNTR, EL, STATUS)
```

In the latter case, HDS will determine the number of characters actually required to format the object's values without loss of information. It uses decimal strings for numerical values and the values 'TRUE' and 'FALSE' to represent logical values as character strings. If the object is already of character type, then its actual length will be used directly. The routine **DAT_MAPC** also operates in this manner.

You should consult SUN/92 for details of how to use this facility on different machines. It is one of the areas where it is very difficult to produce a mechanism which works properly on all machines.

15.1.6 Copying and deleting objects

HDS can also copy and delete objects. Routines **DAT_COPY** and **DAT_ERASE** will recursively copy and erase all levels of the hierarchy below that specified in the subroutine call:

```
      CHARACTER*(DAT__SZLOC) OLOC

*  Copy the AXIS structure to component AXISCOPY of the structure located
*  by OLOC (which must have been previously defined).
      CALL DAT_FIND(NLOC, 'AXIS', ALOC, STATUS)
      CALL DAT_COPY(ALOC, OLOC, 'AXISCOPY', STATUS)
      CALL DAT_ANNUL(ALOC, STATUS)

*  Erase the original AXIS structure.
      CALL DAT_ERASE(NLOC, 'AXIS', STATUS)
```

Note that the locator to the AXIS object has been annulled before attempting to delete it. This whole operation can also be done using **DAT_MOVE**:

```
      CALL DAT_MOVE(ALOC, OLOC, 'AXISCOPY', STATUS)
```

15.1.7 Subsets of objects

The routine **DAT_CELL** accesses a single element of an array. An example was shown in Section 15.1.2. The routine **DAT_SLICE** accesses a subset of an arbitrarily dimensioned object. This subset can then be treated as if it were an object in its own right. For example:

```
      CHARACTER*(DAT__SZLOC) SLICE
      INTEGER LOWER(2), UPPER(2)
      DATA LOWER / 100, 100 /
      DATA UPPER / 200, 200 /

*  Get a locator to the subset DATA_ARRAY(100:200,100:200).
      CALL DAT_FIND(NLOC, 'DATA_ARRAY', LOC, STATUS)
      CALL DAT_SLICE(LOC, 2, LOWER, UPPER, SLICE, STATUS)

*  Map the subset as a vector.
      CALL DAT_MAPV(SLICE, '_REAL', 'UPDATE', PNTR, EL, STATUS)
```

In contrast to **DAT_SLICE**, routine **DAT_ALTER** makes a permanent change to a non-scalar object. The object can be made larger or smaller, but only in the last dimension. This function is entirely dynamic, *i.e.* it can be done at any time, provided the object is not mapped for access. Note that **DAT_ALTER** works on both primitives and structures. It is important to realise that the number of dimensions cannot be changed by **DAT_ALTER**.

15.1.8 Temporary objects

Temporary objects of any type and shape may be created by using the **DAT_TEMP** routine. This returns a locator to the newly created object, and this may then be manipulated just as if it were an ordinary object (in fact a temporary container file is created with a unique name to hold all such objects, and this is deleted when **HDS_STOP** is executed at the end of the program). This is often useful for providing workspace for algorithms which may have to deal with large arrays.

15.1.9 Enquiries

One of the most important properties of HDS is that its data files are self-describing. This means that each object carries with it information describing all its attributes (not just its value), and these attributes can be obtained by means of enquiry routines. An example will illustrate:

```
      PARAMETER (MAXCMP=10)
      CHARACTER*(DAT__SZNAM) NAME(MAXCMP)
      CHARACTER*(DAT__SZTYP) TYPE(MAXCMP)
      INTEGER NCOMP, I
      LOGICAL PRIM(MAXCMP)

*  Enquire the names and types of up to MAXCMP components...

*  First get the total number of components.
      CALL DAT_NCOMP(NLOC, NCOMP, STATUS)

*  Now index through the structure's components, obtaining locators and the
*  required information.
      DO 1 I = 1, MIN(NCOMP,MAXCMP)

*  Get a locator to the I'th component.
         CALL DAT_INDEX(NLOC, I, LOC, STATUS)
```

```
*  Obtain its name and type.
         CALL DAT_NAME(LOC, NAME(I), STATUS)
         CALL DAT_TYPE(LOC, TYPE(I), STATUS)

*  Is it primitive?
         CALL DAT_PRIM(LOC, PRIM(I), STATUS)
         CALL DAT_ANNUL(LOC, STATUS)
 1    CONTINUE
```

Here, **DAT_INDEX** is used to get locators to objects about which (in principle) we know nothing. This is just like listing the files in a directory, except that the order in which the components are stored in an HDS structure is arbitrary (so they won't necessarily be accessed in alphabetical order).

15.1.10 Packaged routines

HDS includes families of routines which provide a more convenient method of accessing objects than the basic routines. For instance, members of the family **DAT_PUT**∗ write values of specific type and dimensionality, and the **DAT_GET**∗ routines read similar values. Thus **DAT_PUT0I** will write a single INTEGER value to a scalar primitive, and **DAT_GET1R** will read the value of a vector primitive and store it in a REAL program array. There are no **DAT_GET2x** routines; all dimensionalities higher than one are handled by **DAT_GETNx** and **DAT_PUTNx**.

Another family of routines are the CMP routines. These access components of the 'current level'. This usually involves:

- FINDing the required object and getting a locator to it.
- Performing the required operation, *e.g.* PUTting some value into it.
- ANNULling the locator.

The CMP routines package this sort of operation, replacing three or so subroutine calls with one. The naming scheme is based on the associated DAT routines. An example is shown below.

```
      CHARACTER*80 DLAB
      INTEGER DIMS(2)
      REAL IMVALS(512, 1024)
      DATA DIMS / 512, 1024 /

*  Get REAL values from the DATA_ARRAY component.
      CALL CMP_GETNR(NLOC, 'DATA_ARRAY', 2, DIMS, IMVALS, DIMS, STATUS)

*  Get a character string from the LABEL component and store it in DLAB.
      CALL CMP_GET0C(NLOC, 'LABEL', DLAB, STATUS)
```

15.2 REF — References to HDS objects

Reference objects are HDS objects which store references to other HDS data objects. They act as pointers to data, rather than storing data themselves.

The REF package allows reference objects to be created and written, and it allows locators to referenced objects to be obtained. The routines are listed in Section 21.2.3.

The referenced object may be defined as *internal*, in which case it is assumed to be within the same container file as the reference object itself, even if the reference object is copied to another container file. In that case the reference must point to an object which has the same pathname within the new file as it had in the old one. References which are not *internal* will point to a named container file.

Reference objects may be copied and erased using DAT_COPY and DAT_ERASE. Care must be taken when copying reference objects or referenced objects, otherwise the reference may no longer point to the referenced object.

Referenced objects must exist at the time the reference is made or used.

15.2.1 Uses

Two main uses for this package are foreseen:

- To maintain a catalogue of data objects.
- To avoid duplicating a large data object.

As an example of the second use, suppose that a large object is logically required to form part of a number of other objects. To avoid duplicating the common object, the others may contain a reference to it. For example:

```
   Name               type                 Comments

DATA               DATA_SETS
  .SET1            SPECTRUM
     .AXIS1        _REAL(1024)         Actual axis data
     .DATA_ARRAY   _REAL(1024)
  .SET2            SPECTRUM
     .AXIS1        REFERENCE_OBJ       Reference to DATA.SET1.AXIS1
     .DATA_ARRAY   _REAL(1024)
  .SET3            SPECTRUM
     .AXIS1        REFERENCE_OBJ       Reference to DATA.SET1.AXIS1
     .DATA_ARRAY   _REAL(1024)
```

Then a piece of code which handles structures of type SPECTRUM, which would normally contain the axis data in .AXIS1 (as SET1 does), could be modified as follows to handle an object .AXIS1 containing either the actual axis data or a reference to the object which does contain the actual axis data.

```
*  LOC1 is a locator associated with a SPECTRUM object; obtain locator to AXIS data
      CALL DAT_FIND(LOC1, 'AXIS1', LOC2, STATUS)

*  Modification to allow AXIS1 to be a reference object; check type of object
      CALL DAT_TYPE(LOC2, TYPE, STATUS)
      IF (TYPE .EQ. 'REFERENCE_OBJ') THEN
          CALL REF_GET(LOC2, 'READ', LOC3, STATUS)
          CALL DAT_ANNUL(LOC2, STATUS)
          CALL DAT_CLONE(LOC3, LOC2, STATUS)
          CALL DAT_ANNUL(LOC3, STATUS)
      ENDIF

*  End of modification.  LOC2 now locates the axis information wherever it is.
```

This code has been packaged into the subroutine **REF_FIND** which can be used instead of DAT_FIND in cases where the component requested may be a reference object.

When a locator which has been obtained in this way is finished with, it should be annulled using REF_ANNUL rather than DAT_ANNUL. This is so that, if the locator was obtained via a reference, the HDS_OPEN for the container file may be matched by an HDS_CLOSE. *Note that this should only be done when any other locators derived from the locator to the referenced object are also finished with.*

15.3 NDF — Extensible n-dimensional data format

The programmer's manual (SUN/33) for NDF runs to 199 pages. It is impossible to produce a useful summary of it, and pointless to reproduce the whole of it here. Instead, we give you an overview of how NDFs are related to HDS, what is in an NDF, and the sort of functions provided by the NDF routines. A realistic, fully commented example program is also given, which should at least give you an idea of how the routines are used. The routines are listed in Section 21.2.1.

15.3.1 Relationship with HDS

The NDF file format is based on HDS, and NDF data structures are stored in HDS container files. However, this does not necessarily mean that all applications which can read HDS files can also handle data stored in NDF format.

To understand why, you must appreciate that HDS provides only a rather low-level set of facilities for storing and handling astronomical data. These include the ability to store primitive data objects (such as arrays of numbers, character strings, *etc.*) in a convenient and self-describing way within *container files*. However, the most important aspect of HDS is its ability to group these primitive objects together to build larger, more complex structures. In this respect, HDS can be regarded as a construction kit which higher-level software can use to build even more sophisticated data formats.

The NDF is a higher-level data format which has been built in this way out of the more primitive facilities provided by HDS. Thus, in HDS terms, an NDF is a data structure constructed according to a particular set of conventions to facilitate the storage of typical astronomical data (such as spectra, images, or similar objects of higher dimensionality).

While HDS can be used to access such structures, it does not contain any of the interpretive knowledge needed to assign astronomical meanings to the various components of an NDF, whose details can become quite complicated. In practice, therefore, it is cumbersome to process NDF data structures using HDS directly. Instead, the NDF access routines are provided. These 'know' how NDF data structures are built, so they can hide the details from writers of astronomical applications. This results in a subroutine library which deals in higher-level concepts more closely related to the work which typical astronomical applications need to perform, and which emphasises the data concepts which an NDF is designed to represent, rather than the details of its implementation.

15.3.2 Data format

The simplest way of regarding an NDF is to view it as a collection of those items which might typically be required in an astronomical image or spectrum. The main part is an N-dimensional array of *data* (where N is 1 for a spectrum, 2 for an image, *etc.*), but this may also be accompanied by a number of other items which are conveniently categorised as follows:

Character components:	**TITLE**	— NDF title
	LABEL	— Data label
	UNITS	— Data units
Array components:	**DATA**	— Data pixel values
	VARIANCE	— Pixel variance estimates
	QUALITY	— Pixel quality values
Miscellaneous components:	**AXIS**	— Coordinate axes
	HISTORY	— Processing history[5]
Extensions:	**EXTENSION**	— Provides extensibility

The names of these components are significant, since they are used by the NDF access routines to identify the component(s) to which certain operations should be applied.[6] The following describes the purpose and interpretation of each component in slightly more detail.

Character components:

TITLE — This is a character string whose value is intended for general use as a heading for such things as graphical output; *e.g.* 'M51 in good seeing'.

LABEL — This is a character string whose value is intended to be used on the axis of graphs to describe the quantity in the NDF's *data* component; *e.g.* 'Surface brightness'.

UNITS — This is a character string whose value describes the physical units of the quantity stored in the NDF's *data* component; *e.g.* 'J/(m∗∗2∗Ang∗s)'.

Array components:

DATA — This is an N-dimensional array of pixel values representing the spectrum, image, *etc.* stored in the NDF. This is the only NDF component which must **always** be present. All the others are optional.

VARIANCE — This is an array of the same shape and size as the *data* array, and represents the measurement errors or uncertainties associated with the individual *data* values. If present, these are always stored as *variance* estimates for each pixel.

QUALITY — This is an array of the same shape and size as the *data* array which holds a set of unsigned byte values. These are used to assign additional 'quality' attributes to each pixel (for instance, whether it is part of the active area of a detector). Quality values may be used to influence the way in which the NDF's *data* and *variance* components are processed, both by general-purpose software and by specialised applications.

Miscellaneous components:

AXIS — This represents a group of *axis* components which may be used to describe the shape and position of the NDF's pixels in a rectangular coordinate system. The physical units and a label for each axis of this coordinate system may also be stored. (Note that the ability to associate *extensions* with an NDF's *axis* coordinate system, although described in SGP/38, is not yet available via the NDF access routines.)

[5]The *history* component is not fully supported by the present version of the NDF access routines.

[6]Note that the name 'DATA' used by the NDF_ routines to refer to an NDF's *data* component differs from the actual name of the HDS object in which it is stored, which is 'DATA_ARRAY'.

HISTORY — This may be used to keep a record of the processing history. If present, this component should be updated by any applications which modify the data structure. Support for this component is not yet provided by the NDF access routines.

Extensions:

EXTENSIONs — These are user-defined HDS structures associated with the NDF, and are used to give the data format flexibility by allowing it to be extended. Their formulation is not covered by the NDF definition, but a few simple routines are provided for accessing and manipulating named extensions, and for reading and writing the values of components stored within them.

15.3.3 Routines

The NDF access routines are listed in Section 21.2.1. They perform the following types of operation on NDF data structures:

- Obtaining access, both input and output.
- Creating and deleting.
- Enquiring about attributes, including shape and size.
- Enquiring about attributes of components.
- Reading, writing, and resetting component values.
- Enquiring about (and flagging) the presence of *bad* pixels in components.
- Accessing and handling *quality* information.
- Modifying attributes (including shape and size) and those of components (such as numeric type).
- Reading, writing, and resetting the values of *axis* arrays and other *axis* components.
- Modifying the attributes of *axis* components (such as the numeric type of *axis* arrays).
- Controlling the propagation of components to output data structures.
- Creating, deleting, and enquiring about extensions, and obtaining access to components stored within extensions.
- Controlling the propagation of extensions to output data structures.
- Selecting and managing *sections* which refer to subsets or super-sets.
- Merging attributes to match the processing capabilities of specific applications.
- Importing, finding, and copying objects held in container files.
- Constructing messages.
- Controlling access.

Programming support for these routines, including on-line help, is also provided by the Starlink language-sensitive editor STARLSE.

15.3.4 Example program

The following application adds two NDF data structures pixel-by-pixel. It is a fairly sophisticated 'add' application which will handle both the *data* and *variance* components, as well as coping with NDFs of any shape and data type.

```
      SUBROUTINE ADD(STATUS)
*+
*  Name:
*     ADD

*  Purpose:
*     Add two NDF data structures.

*  Description:
*     This routine adds two NDF data structures pixel-by-pixel to produce a new NDF.

*  ADAM Parameters:
*     IN1 = NDF (Read)
*        First NDF to be added.
*     IN2 = NDF (Read)
*        Second NDF to be added.
*     OUT = NDF (Write)
*        Output NDF to contain the sum of the two input NDFs.
*     TITLE = LITERAL (Read)
*        Value for the title of the output NDF.  A null value will cause
*        the title of the NDF supplied for parameter IN1 to be used instead.
*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants
      INCLUDE 'NDF_PAR'          ! NDF_ public constants

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      CHARACTER*(13) COMP        ! NDF component list
      CHARACTER*(NDF__SZFTP) DTYPE ! Type for output components
      CHARACTER*(NDF__SZTYP) ITYPE ! Numeric type for processing
      INTEGER EL                 ! Number of mapped elements
      INTEGER IERR               ! Position of first error (dummy)
      INTEGER NDF1               ! Identifier for 1st NDF (input)
      INTEGER NDF2               ! Identifier for 2nd NDF (input)
      INTEGER NDF3               ! Identifier for 3rd NDF (output)
      INTEGER NERR               ! Number of errors
      INTEGER PNTR1(2)           ! Pointers to 1st NDF mapped arrays
      INTEGER PNTR2(2)           ! Pointers to 2nd NDF mapped arrays
      INTEGER PNTR3(2)           ! Pointers to 3rd NDF mapped arrays
      LOGICAL BAD                ! Need to check for bad pixels?
      LOGICAL VAR1               ! Variance component in 1st input NDF?
      LOGICAL VAR2               ! Variance component in 2nd input NDF?
*.
*  Check inherited global status.
      IF (STATUS.NE.SAI__OK) RETURN

*  Begin an NDF context.
      CALL NDF_BEGIN

*  Obtain identifiers for the two input NDFs.  These involve calls to the
*  parameter system and may be resolved from the interface file, the command
*  line, or by prompting the user.
```

```
      CALL NDF_ASSOC('IN1', 'READ', NDF1, STATUS)
      CALL NDF_ASSOC('IN2', 'READ', NDF2, STATUS)

*  Trim their pixel-index bounds to match.  This selects the largest common
*  set of pixels from the two input arrays.
      CALL NDF_MBND('TRIM', NDF1, NDF2, STATUS)

*  Create a new output NDF based on the first input NDF.  Propagate the axis
*  and quality components, which are not changed.  This program does not
*  support the units component.
      CALL NDF_PROP(NDF1, 'Axis,Quality', 'OUT', NDF3, STATUS)

*  See if a variance component is available in both input NDFs and generate
*  an appropriate list of input components to be processed.
      CALL NDF_STATE(NDF1, 'Variance', VAR1, STATUS)
      CALL NDF_STATE(NDF2, 'Variance', VAR2, STATUS)
      IF (VAR1.AND.VAR2) THEN
         COMP = 'Data,Variance'
      ELSE
         COMP = 'Data'
      END IF

*  Determine which numeric type to use to process the input arrays and set an
*  appropriate type for the corresponding output arrays.  This program supports
*  integer, real and double-precision arithmetic.  ITYPE says what type should
*  be used for the processing;  DTYPE is the type needed for the output data
*  (identified by NDF3) and so is passed on to NDF_STYPE
      CALL NDF_MTYPE('_INTEGER,_REAL,_DOUBLE',
     :                NDF1, NDF2, COMP, ITYPE, DTYPE, STATUS)
      CALL NDF_STYPE(DTYPE, NDF3, COMP, STATUS)

*  Map the input and output arrays.  Note that the identifier NDF3 produced by
*  NDF_PROP is used for the output data, which must be in WRITE access mode.
      CALL NDF_MAP(NDF1, COMP, ITYPE, 'READ', PNTR1, EL, STATUS)
      CALL NDF_MAP(NDF2, COMP, ITYPE, 'READ', PNTR2, EL, STATUS)
      CALL NDF_MAP(NDF3, COMP, ITYPE, 'WRITE', PNTR3, EL, STATUS)

*  Merge the bad pixel flag values for the input data arrays to see if checks
*  for bad pixels are needed.  The first argument `.TRUE.' says that this
*  application can handle bad pixels (if it were .FALSE. and bad pixels were
*  present the STATUS would be set to an error value).  The fifth argument
*  `.FALSE.' says not to check whether there actually are any bad pixels present.
      CALL NDF_MBAD(.TRUE., NDF1, NDF2, 'Data', .FALSE., BAD, STATUS)

*  Select the appropriate routine for the data type being processed and add the data
*  arrays.  Note that the arithmetic is done by one of the VEC_ routines in PRIMDAT
*  which handle bad pixels and any arithmetic errors, such as overflow, for you.
      IF (STATUS.EQ.SAI__OK) THEN
         IF (ITYPE.EQ.'_INTEGER') THEN
            CALL VEC_ADDI(BAD, EL, %VAL(PNTR1(1)),
     :                    %VAL(PNTR2(1)), %VAL(PNTR3(1)),
     :                    IERR, NERR, STATUS)

         ELSE IF (ITYPE.EQ.'_REAL') THEN
            CALL VEC_ADDR(BAD, EL, %VAL(PNTR1(1)),
     :                    %VAL(PNTR2(1)), %VAL(PNTR3(1)),
     :                    IERR, NERR, STATUS)

         ELSE IF (ITYPE.EQ.'_DOUBLE') THEN
```

```
            CALL VEC_ADDD(BAD, EL, %VAL(PNTR1(1)),
     :                    %VAL(PNTR2(1)), %VAL(PNTR3(1)),
     :                    IERR, NERR, STATUS)
         END IF

*  Flush any messages resulting from numerical errors.
         IF (STATUS.NE.SAI__OK) CALL ERR_FLUSH(STATUS)
      END IF

*  See if there may be bad pixels in the output data array and set the output
*  bad pixel flag value accordingly.  NERR is the number of errors detected by
*  the VEC_ADDx routine.
      BAD = BAD .OR. (NERR.NE.0)
      CALL NDF_SBAD(BAD, NDF3, 'Data', STATUS)

*  If variance arrays are also to be processed (i.e. added), then see if bad
*  pixels may be present in the variance arrays.
      IF (VAR1.AND.VAR2) THEN
         CALL NDF_MBAD(.TRUE., NDF1, NDF2, 'Variance', .FALSE., BAD,
     :                 STATUS)

*  Select the appropriate routine to add the variance arrays.
         IF (STATUS.EQ.SAI__OK) THEN
            IF (ITYPE.EQ.'_INTEGER') THEN
               CALL VEC_ADDI(BAD, EL, %VAL(PNTR1(2)),
     :                       %VAL(PNTR2(2)), %VAL(PNTR3(2)),
     :                       IERR, NERR, STATUS)

            ELSE IF (ITYPE.EQ.'_REAL') THEN
               CALL VEC_ADDR(BAD, EL, %VAL(PNTR1(2)),
     :                       %VAL(PNTR2(2)), %VAL(PNTR3(2)),
     :                       IERR, NERR, STATUS)

            ELSE IF (ITYPE.EQ.'_DOUBLE') THEN
               CALL VEC_ADDD(BAD, EL, %VAL(PNTR1(2)),
     :                       %VAL(PNTR2(2)), %VAL(PNTR3(2)),
     :                       IERR, NERR, STATUS)
            END IF

*  Flush any messages resulting from numerical errors.
            IF (STATUS.NE.SAI__OK) CALL ERR_FLUSH(STATUS)
         END IF

*  See if bad pixels may be present in the output variance array and set the
*  bad pixel flag accordingly.
         BAD = BAD .OR. (NERR.NE.0)
         CALL NDF_SBAD(BAD, NDF3, 'Variance', STATUS)
      END IF

*  Obtain a new title for the output NDF, by way of the parameter system.
      CALL NDF_CINP('TITLE', NDF3, 'Title', STATUS)

*  End the NDF context.
      CALL NDF_END(STATUS)

*  If an error occurred, then report context information.
      IF (STATUS.NE.SAI__OK) THEN
         CALL ERR_REP('ADD_ERR',
     :   'ADD: Error adding two NDF data structures.', STATUS)
```

```
      END IF

      END
```

The following is a possible interface file for the above application:

```
interface ADD
   parameter IN1                 # First input NDF
      position 1
      prompt   'First input NDF'
   endparameter
   parameter IN2                 # Second input NDF
      position 2
      prompt   'Second input NDF'
   endparameter
   parameter OUT                 # Output NDF
      position 3
      prompt   'Output NDF'
   endparameter
   parameter TITLE               # Title for output NDF
      type     'LITERAL'
      prompt   'Title for output NDF'
      vpath    'DEFAULT'
      default  !
   endparameter
endinterface
```

Chapter 16
The Message and Error Systems

There is a general need for application programs to provide the user with informative messages about:

- What they do — for example, during long operations it is helpful if the user is kept informed of what a program is doing.
- What results have been obtained — for example, the notification of the final results from a procedure, or of some intermediate results that would help the user respond to further prompts.
- What errors have occurred — for example, errors which lead to the user being prompted to provide more sensible input to a program, or fatal errors which cause an application to stop.

This chapter describes two subroutine libraries which can be used for this purpose. They are:

MSG — Message Reporting System.
ERR — Error Reporting System.

These are fully described in SUN/104.

16.1 MSG — Message reporting system

The obvious way to produce messages in Fortran programs is by WRITE and PRINT statements. It is possible to construct the text of a message from various components, including numbers, formatted in a CHARACTER variable using the internal WRITE statement. The resulting message may then be displayed. However, it is sometimes difficult to format numerical output in its most concise form. To do this in-line each time a message is sent to the user would be very inconvenient and justifies the provision of a dedicated set of subroutines. These considerations led to the production Message Reporting System.

16.1.1 Reporting messages

The primary message reporting subroutine is MSG_OUT. It has a calling sequence of the form:

```
      CALL MSG_OUT(PARAM, TEXT, STATUS)
```

where PARAM is a character string giving the name of the message, TEXT is a character string giving the message text, and STATUS is the integer *global status* value.

It sends the message string, TEXT, to the standard output stream. This will normally be the user's terminal, but is the log file for a batch job. The maximum message length is 200 characters. If it exceeds this, it is truncated with an ellipsis, *i.e.* '. . . ', but no error results.

Here is an example of using MSG_OUT:

```
      CALL MSG_OUT('EXAMPLE_MSGOUT', 'An example of MSG_OUT.', STATUS)
```

It is sometimes useful to add blank lines for clarity. MSG_BLANK does this:

```
      CALL MSG_BLANK(STATUS)
```

Messages can also be stored in a character variable, rather than being output. MSG_LOAD does this.

16.1.2 Conditional message reporting

It is sometimes useful to have varying levels of message output which may be controlled by the user. This is achieved by MSG_OUTIF:

```
      CALL MSG_OUTIF(MSG__NORM, 'MESS_NAME', 'A conditional message', STATUS)
```

Here, the first argument is the 'priority' associated with the message. It has three possible values, which represent filter levels:

MSG__QUIET — always output message, regardless of output filter setting.

MSG__NORM — output message if current output filter is set to either MSG__NORM or MSG__VERB.

MSG__VERB — output message only if current output filter is set to MSG__VERB.

(I know this looks wrong on a first reading, but 'quiet' messages are messages that are output in 'quiet' mode, and are therefore the loudest!) The default is MSG__NORM. It may be modified by MSG_IFSET, for example:

```
      CALL MSG_IFSET(MSG__QUIET, STATUS)
```

16.1.3 Message tokens

Applications often need to include the values of Fortran variables in output messages. This is done in the Message System using tokens embedded in the message text. For example, a program which measures the intensity of an emission line in a spectrum can output its result by:

```
      CALL MSG_SETR('FLUX', FLUX)
      CALL MSG_OUT('EXAMPLE_RESULT',
     :             'Emission flux is ^FLUX (erg/cm2/A/s).', STATUS)
```

Here, MSG_SETR assigns the result, stored in the REAL variable FLUX, to the message token named 'FLUX'. The token string is then included in the message text by prefixing it with the up-arrow, '^', escape character. The usual set of similar routines is provided to handle the other standard data types.

An additional feature of the MSG_SETx routines is that calls using an existing token name will result in the value being appended to any previously assigned token string. Here is the previous example written to exploit this feature:

```
      FUNITS = ' erg/cm2/A/s'
      CALL MSG_SETR('FLUX', FLUX)
      CALL MSG_SETC('FLUX', FUNITS)
      CALL MSG_OUT('EXAMPLE_RESULT',
     :             'Emission flux is ^FLUX.', STATUS)
```

Note that repeated calls append values with no separator, and hence a leading space is needed in the FUNITS string to separate the flux value and its units in the expanded message.

Sometimes a specific format is required; for example, the message may form part of a table. MSG_FMTx is a set of subroutines of the form:

```
      CALL MSG_FMTx(TOKEN, FORMAT, VALUE)
```

where FORMAT is a valid Fortran 77 format string which can be used to encode the supplied value, VALUE. As for MSG_SETx, *x* corresponds to each of the five standard Fortran 77 data types — *D*, *R*, *I*, *L* and *C*.

Sometimes it is necessary to include the message token escape character, '^', literally in a message. When the character is not the last in a message string, it can be included literally by duplicating it. When it is immediately followed by a blank, or is at the end of the MSG_OUT text, it is included literally. Escape characters and token names will also be output literally if they appear within the value assigned to a message token; *i.e.* message token substitution is not recursive.

16.1.4 Message parameters

In calls to MSG_OUT and MSG_LOAD, the message name is the name of a message parameter which is associated with the message text. This name should be no more than 15 characters long and may be associated with a message specified in the interface file. When the message parameter is specified in the interface file, this text is used in preference to that given in the argument list.

Here is an example of using MSG_OUT:

```
      CALL MSG_OUT('RD_TAPE', 'Reading tape.', STATUS)
```

This will generate the message:

```
    Reading tape.
```

If the message parameter, 'RD_TAPE', is associated with a different text string in the interface file, *e.g.*

```
    message RD_TAPE
       text 'The program is currently reading the tape, please wait.'
    endmessage
```

then the output message would be the one defined in the interface file:

```
    The program is currently reading the tape, please wait.
```

This enables ADAM applications to support foreign languages.

16.1.5 Parameter references

We may need to refer to a program parameter in a message. There are two kinds of reference required:

- The *keyword*.
- The *name* of an object, device, or file.

We can include references of these kinds by prefixing parameter names with an *escape character* of which there are three: '^', '%', '$'. To include a parameter's keyword, prefix its name with a '%' character, as in:

```
CALL MSG_OUT('ET_RANGE', '%ET parameter is ignored', STATUS)
```

If the keyword of parameter ET is 'EXPOSURE_TIME', the output generated by this call is:

```
EXPOSURE_TIME parameter is ignored
```

To include the name of an object, device, or file associated with a parameter, prefix its name with a '$' character, as in:

```
CALL MSG_OUT('CREATING', 'Creating $DATASET', STATUS)
```

If parameter DATASET is associated with an object called SWP1234, this would produce the output:

```
Creating SWP1234
```

Sometimes a parameter name is contained in a variable. In order to use it in a message, a message token can be associated with its name. For example:

```
  CALL MSG_SETC('PAR', PNAME)
  CALL MSG_OUT('PAR_UNEXP', '%^PAR=0.0 unexpected', STATUS)
```

In this example, the escape sequence '^' prefixes the token name, 'PAR'. The sequence '^PAR' gets replaced by the contents of the character string contained in the variable PNAME; this, being prefixed by '%', then gets replaced in the final message by the keyword associated with the parameter.

16.1.6 Getting the conditional output level

Routine MSG_IFSET sets the filter level for conditional message output. Routine MSG_IFGET also sets it, but gets its value from the parameter system:

```
  CALL MSG_IFGET(PNAME, STATUS)
```

where PNAME is the parameter name. Always use the same name, MSG_FILTER, for this purpose. The three acceptable strings are:

QUIET — representing MSG__QUIET;

NORMAL — representing MSG__NORM;

VERBOSE — representing MSG__VERB.

Any other value will result in an error report and the status value being set to MSG__INVIF.

16.2 ERR — Error reporting system

Although the Message System could be used for reporting errors, there are a number of reasons why a separate facility should be provided:

- The Message System uses the inherited status scheme, so MSG_OUT and MSG_LOAD will not execute if STATUS has an error value. Consequently, it cannot report information about an earlier error.
- In a program consisting of many subroutine levels, each routine which has something informative to say about an error should be able to contribute to the information that the user receives. This includes:
  - The routine which first detects the error, as this will probably have access to information which is hidden from higher level routines.
  - The chain of routines between the main program and the routine in which the error originated. Some of these will usually be able to report on the context in which the error occurred, and so add relevant information which is not available to routines at lower levels.

  This can lead to several error reports arising from a single failure.
- It is not always necessary for an error report to reach the user. For example, a routine may decide that it can safely handle an error detected at a lower level without informing the user. In this case, error reports associated with the error should be discarded, and this can only happen if the output of error messages to the user is deferred.

These considerations have led to the design and implementation of a set of routines which form the Error Reporting System.

16.2.1 Inherited status checking

The recommended method of indicating when errors have occurred in Starlink software is to use an integer status value in each subroutine argument list. This inherited status argument, say STATUS, should always be the last argument and every routine should check its value on entry. The *ADAM Error Strategy* is as follows:

- The routine returns without action if STATUS is input with a value other than SAI__OK.
- The routine leaves STATUS unchanged if it completes successfully.
- The routine sets STATUS to an appropriate error value and outputs an error message if it fails to complete successfully.

Note that it is often useful to use a status argument and inherited status checking in routines which 'cannot fail'. This prevents them executing, possibly producing a run-time error, if their arguments contain rubbish after a previous error. Every piece of software that calls such a routine is then saved from making an extra status check. Furthermore, if the routine is later upgraded it may acquire the potential to fail, and so a status argument will subsequently be required. If a status argument is included initially, existing code which calls the routine will not need changing.

16.2.2 Reporting errors

The routine used to report errors is ERR_REP. It has a calling sequence of the form:

```
      CALL ERR_REP(PARAM, TEXT, STATUS)
```

where PARAM is the error message name, TEXT is the error message text, and STATUS is the inherited status. These arguments are similar to those used in the Message System routine MSG_OUT.

The error message name, PARAM, should be a globally unique identifier for the error report with the form:

```
    routn_message
```

for routines in an application, or:

```
    fac_routn_message
```

for routines in a subroutine library. In the former case, `routn` is the name of the application routine from which ERR_REP is being called, and `message` is a sequence of characters uniquely identifying the error report within that routine. In the latter case, `fac_routn` is the full name of the routine from which ERR_REP is being called, and `message` is a sequence of characters unique within that routine. These naming conventions are designed to ensure that each error report made within a complete software system has a unique error name associated with it.

Here is a simple example of error reporting where part of the application code detects an invalid value of some kind, sets STATUS, reports the error, and then aborts:

```
      IF (<invalid value>) THEN
         STATUS = SAI__ERROR
         CALL ERR_REP('ROUTN_BADV', 'Value is invalid.', STATUS)
         GO TO 999
      END IF
      ...
 999  CONTINUE
      END
```

This sequence of three operations:

(1) Set STATUS to an error value.
(2) Report an error.
(3) Abort.

is the standard response to an error condition and should be adopted by all software which uses the Error System.

Note that ERR_REP differs from MSG_OUT in that ERR_REP will execute regardless of the input value of STATUS. Although the Starlink convention is for routines not to execute if their status argument indicates a previous error, the Error System routines obviously cannot behave in this way if their purpose is to report these errors.

Message tokens can be used in ERR_REP in the same way as in MSG_OUT.

16.2.3 When to report an error

In the following example, part of an application makes a series of routine calls:

```
      CALL ROUTN1(A, B, STATUS)
      CALL ROUTN2(C, STATUS)
      CALL ROUTN3(T, Z, STATUS)

*  Check the global status.
      IF (STATUS .NE. SAI__OK) GO TO 999
      .
999   CONTINUE
      END
```

Each routine uses the inherited status strategy and reports errors by calling ERR_REP. If an error occurs within any of them, STATUS will be set to an error value and inherited status checking by all subsequent routines will cause them not to execute. Thus, it becomes unnecessary to check for an error after each routine call, and a single check at the end of the sequence of calls is all that is required to handle correctly any error condition that may arise. Because an error report will already have been made by the routine that failed, it is usually sufficient simply to abort if an error arises in a sequence of routine calls.

It is important to distinguish the case where a called subroutine sets STATUS and makes its own error report, as above, from the case where STATUS is set explicitly as a result of a directly detected error, as in the previous example. If the error reporting strategy is to function correctly, then responsibility for reporting the error must lie with the routine which modifies the status argument. The golden rule is therefore:

If STATUS is explicitly set to an error value, then an accompanying call to ERR_REP **must** *be made.*

Unless there are good documented reasons why this cannot be done, routines which return a bad status value and do not make an accompanying error report should be regarded as containing a bug[1].

16.2.4 Setting and defining status values

Normally, set status values to the global constants SAI__OK and SAI__ERROR. However, when writing subroutine libraries it is useful to have a larger number of error codes available and to define these in a separate include file. The naming convention:

```
fac__ecode
```

should be used for the names of error codes where `fac` is the three-character facility prefix and `ecode` is up to five alphanumeric characters of error code name. *Note the double underscore used in this naming convention.* The include file should be referred to by the name fac_ERR, *e.g.*

```
      INCLUDE 'SGS_ERR'
```

where in this case the facility name is `SGS` (Simple Graphics System). These symbolic constants should be defined at the beginning of every routine which requires them, prior to the declaration of any subroutine arguments or local variables.

[1]For historical reasons there are still many routines in ADAM which set a status value without making an accompanying error report — these are gradually being corrected. If such a routine is used before it has been corrected, then the strategy outlined here is recommended. It is advisable not to complicate new code by attempting to make an error report on behalf of the faulty subroutine. If appropriate, please tell the relevant support person about the problem.

The purpose of error codes is to enable the status argument to indicate that an error has occurred by having a value which is not equal to SAI__OK. By using a set of pre-defined error codes the calling routine is able to test the returned status to distinguish between error conditions which may require different action. Generally, it is not necessary to define a large number of error codes which would allow a unique value to be used every time an error report is made. It is sufficient to be able to distinguish the important classes of error which may occur. Examples of existing software can be consulted as a guide in this matter.

Software from outside a package which defines a set of error codes may use that package's codes to test for specific error conditions arising within that package. However, with the exception of the SAI__ codes, it should *not* assign these values to the status argument. To do so could cause confusion about which package detected the error.

16.2.5 The content of error messages

The purpose of an error message is to be informative and it should therefore provide as much relevant information about the context of the error as possible. It should not be misleading or contain irrelevant information. Particular care is necessary when reporting errors from routines which might be called by a wide variety of software. They should not make unjustified assumptions. For example, in a routine that adds two arrays, the report:

```
!! Error adding two arrays.
```

would be preferable to:

```
!! Error adding two images.
```

if the same routine could be called to add two spectra!

Normally it is adequate to report an error when is first detected, followed by a further report from the 'top-level' routine. Only include routine names in error reports if they appear in documentation.

16.2.6 Deferred error reporting

The Error System can defer the output of a message to the user, and this allows the final delivery of error messages to be controlled. This is done by the routines ERR_MARK, ERR_RLSE, ERR_FLUSH and ERR_ANNUL. This section describes their functions and how they are used.

The purpose of deferred error reporting can be illustrated by the following example. Consider a routine, HELPER, which detects an error during execution. It reports the error to its caller, giving as much contextual information as it can. It also returns an error status, enabling the caller to react appropriately. However, what may be considered an 'error' in HELPER, *e.g.* an 'end of file' condition, may be considered by the caller to be something that can be handled without informing the user, *e.g.* by terminating its input. Thus, although HELPER will always report the error, it is not always necessary for the associated error message to reach the user. Deferred error reporting enables programs to handle such errors internally.

Suppose HELPER reports an error. At this point, the error message may, or may not, have been received by the user — this will depend on the environment and on whether the caller deferred the error report. HELPER should not ensure delivery of the message to the user; its responsibility ends when it aborts, and responsibility for handling the error condition passes to the caller.

Suppose HELPER is called by HELPED which defers error messages so it can decide how to handle errors. It does this by calling ERR_MARK before HELPER. This ensures that subsequent error messages are deferred and stored in an 'error table'. It also starts a new 'error context' which is independent of previous error messages or tokens. Routine ERR_RLSE returns to the previous context, whereupon any messages in the new error context are transferred to the previous context. In this way, no existing error messages can be lost through deferral. ERR_MARK and ERR_RLSE should always be called in matching pairs and can be nested.

After deferring the error messages, HELPED can handle the error condition in one of two ways:

- by calling ERR_ANNUL, which 'annuls' the error, deleting any deferred error messages in the current context and resetting STATUS to SAI__OK. This causes the error condition to be ignored.
- by calling ERR_FLUSH, which 'flushes out' the error, sending any deferred error messages in the current context to the user and resetting STATUS to SAI__OK. This notifies the user that a problem has occurred, but allows the application to continue anyway.

When an application finally ends, the value of the status argument will reflect whether or not it finished with an error condition. At this point, any remaining error messages will be delivered automatically to the user.

16.2.7 Error message parameters

In calls to ERR_REP, the error name is the name of a message parameter which is associated with the error message text. Like the message parameters used in MSG_OUT and MSG_LOAD, those used in ERR_REP may be associated with an error message specified in the interface file as well as in the argument list.

Chapter 17 The Graphics System

The Starlink graphics software was introduced in Chapter 13. A guide to its use is available in SUN/113. The main packages are:

PGPLOT — is recommended for general use. It provides an easy to use yet powerful way of plotting 2-d axes, contour maps, grey-scale, colour images, and many other styles of plot. It is designed to appeal to astronomers and is especially good at publication quality laser printer output. It cannot be used in conjunction with SGS or any other plotting package.

NCAR/SNX — provides similar facilities to PGPLOT plus some additional styles of plot (e.g. stream-line diagrams) and in general is more powerful (for example, it can draw an X,Y plot with different scales on the right and left hand axes) but is somewhat harder to use. It also contains a 3-d drawing package. You can call SGS and GKS by using SNX.

NAG — graphics can also be used, but it is not encouraged for software that is to become part of the Starlink software collection because its availability on all hardware platforms is not guaranteed (particulary in its single precision form) and non-Starlink sites may not have a licence to use it. NCAR and NAG graphics do not have their own mechanisms for opening graphics devices and SGS or GKS must be used to open a device before plotting can begin.

SGS — should be used for line graphics which requires a style not catered for by PGPLOT or NCAR. This is "lower level" than PGPLOT or NCAR (there are no facilities for drawing axes for example) but it does have excellent facilities for manipulating transformations and "organising" the display surface. SGS programs can also make direct calls to GKS (with a few restrictions), so the full power of GKS is available if SGS's facilities are not adequate.

GKS — should normally not be used because getting even simple things done (e.g. opening a workstation) can be very long-winded and more easily achieved with SGS. Furthermore, achieving device independence is far from straight-forward; all the necessary facilities are available but making proper use of them requires an in-depth knowledge of GKS. Again, this is better left to SGS or higher level packages.

Three more specialised graphics facilities provided by ADAM are:

IDI — should be used in programs requiring intimate interaction with an image display device. This is not based on GKS and offers access to facilities that are outside the GKS model of a graphics device. Currently, the only devices supported are the Ikon image display (on VAXs only) and X-windows.

AGI — should be used where a suite of graphics programs are being written, or where a program is to interact with graphics produced by another package (e.g. KAPPA). It provides facilities for sharing information about pictures such as extent and world coordinates.

GNS — enables a program to obtain information about graphics devices.

Before proceeding, a word of warning which is applicable to all Starlink graphics. Do *not* examine the workstation type or name in order to control the behaviour of a program. For example, you are not allowed to say 'the GKS workstation type is 3200, therefore this is an Ikon image display, therefore colour is available'. This information must be obtained by calling the appropriate GKS, SGS, or PGPLOT inquiry routines; in the unlikely event of the property you are interested in not being available through such inquiries, then devise some mechanism external to the program, or solicit information from the user.

17.1 SGS — Simple graphics system

The following example graphics program uses SGS to draw a circle on a selected graphics device:

```
      SUBROUTINE CIRCLE(STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INTEGER STATUS
      REAL RADIUS           ! Radius of circle
      INTEGER ZONE          ! SGS Zone
*.........................................................
      IF (STATUS.NE.SAI__OK) RETURN
      CALL PAR_GETOR('RADIUS', RADIUS, STATUS)
      CALL SGS_ASSOC('DEVICE', 'WRITE', ZONE, STATUS)
      IF (STATUS.EQ.SAI__OK) THEN
          CALL SGS_CIRCL(0.5, 0.5, RADIUS)
      END IF
      CALL SGS_ANNUL(ZONE, STATUS)
      CALL SGS_DEACT(STATUS)
      END
```

The statements concerned specifically with graphics will now be considered individually.

```
      CALL SGS_ASSOC('DEVICE', 'WRITE', ZONE, STATUS)
```

This makes the association between the parameter DEVICE and an SGS zone on the graphics device provided by the parameter system. The zone identifier is returned to the program in the ZONE variable. SGS and the zone are made ready for graphical output. The second parameter is the access mode whose possible values are 'READ', 'WRITE', and 'UPDATE'. The value 'WRITE' will cause the screen to be cleared; the other values cause the screen to be left as it is.

```
      CALL SGS_CIRCL(0.5, 0.5, RADIUS)
```

This draws a circle centered at (0.5,0.5) in world coordinates with radius RADIUS.

```
      CALL SGS_ANNUL(ZONE, STATUS)
```

This closes the graphics workstation without cancelling the associated parameter. The routine SGS_CANCL is similar but *does* cancel the parameter.

```
      CALL SGS_DEACT(STATUS)
```

This de-activates SGS.

A possible interface file for this program is:

```
interface CIRCLE
   parameter RADIUS
      type     '_REAL'
      position 1
      prompt   'Radius of Circle'
      ppath    'current'
      vpath    'prompt'
   endparameter
   parameter DEVICE
      ptype    'DEVICE'
      position 2
      prompt   'Graphics Device'
      ppath    'current'
      vpath    'prompt'
   endparameter
endinterface
```

After compiling and linking CIRCLE:

```
$ for circle
$ alink circle
```

we can run it from ICL. In the following example, circles of different radii are plotted on different devices (things like device names and print queues are likely to be different for you):

```
ICL> define circle circle
ICL> circle 0.3 graphon
ICL> circle
RADIUS - Radius of Circle /0.3/ > 0.2
DEVICE - Graphics Device /@graphon/ > postscript_l
ICL> $ print/q=sys_ps/form=post gks_72.ps
ICL>
```

With hard copy graphics devices, such as 'postscript_l', a file is created which must then be sent to the device with a DCL PRINT command.

17.2 IDI — Image display interface

IDI is a standard for displaying astronomical data on an image display. It is introduced in SUN/65, and the routine specifications are given in 'An image display interface for astronomical image processing' by D L Terrett, et al, published in *Astron. Astrophys. Suppl. Ser.* **76**, *263-304 (1988).* It has been integrated with ADAM by the usual ASSOC, ANNUL, and CANCL routines discussed in Chapter 14. An example of its use is contained in SUN/65.

IDI is intended for programs that need to manipulate images to a greater extent than can be done with GKS and its offspring. Thus, it does not supersede GKS, but offers features not available in GKS. It is not as good as GKS for producing line plots or character annotation — it does have routines to draw lines and plot text, but these are primitive and offer you little control over how the result will appear in terms of character sizes, style of line widths etc.

Its major strength is its ability to perform many types of interaction using the mouse. Like GKS, it can display an image, move the cursor, and rotate the look-up table. However, it can also scroll and zoom, blink, and read back a representation of the whole display, which can then be used to obtain a hardcopy.

IDI allows these functions to be programmed in a device independent way, so a program can use any device for which IDI has been implemented.

It is not possible to mix GKS and IDI calls on the same display; the two packages use completely different display models. However, it is possible to run the packages one after the other. For example, IDI could follow GKS and the display could be opened without resetting it. However, this is an undesirable approach since the results could be unpredictable. A better solution is to use AGI to mediate between the packages.

17.3 AGI — Applications graphics interface

One of the recurring problems in graphics programming is that you would like a program to know about graphics drawn by another program. For instance, you display an image of the sky, then later you want to obtain the co-ordinates of the stars within the image via the cursor. There are two ways of doing this. The first is to duplicate the display code in the cursor application, which is wasteful and inflexible. The second is to store information about *pictures* in a database that can be accessed by graphics programs. Immediately after a picture is created, information about its position and extent can be added to the database. This second approach is implemented by AGI which can store information about plots on any graphics device.

The AGI graphics database is stored in file AGI_USER:AGI_<hostname>.SDF which contains a data object named WORKSTATION comprising a set of components, one set for each device used. Each type of workstation used causes the generation of a WORKSTATION structure in the database (which is specific to each user). The PICTURE structure stores details of one or more 'pictures' which are visible on the workstation's display. The AGI routines are used to create and manipulate these picture structures in the database. The AGI documentation (SUN/48) talks about 'storing pictures' in the database. Don't be alarmed by this loose terminology; the picture itself isn't stored, just the following information:

PNAME — A generic name specifying the type of the picture. Currently the following names are used:

- *BASE* indicates the base picture — the picture that extends over the whole of the plotting area. This name is reserved and should not be used to name other pictures.
- *DATA* indicates that the picture contains a representation of data in a graphical form, e.g. a greyscale plot or a histogram.
- *FRAME* indicates collections of DATA pictures. For instance, in a contour plot, the DATA picture is the area where contours may potentially be drawn, whereas the FRAME picture comprises the annotated axes and labels, the key, and the data area.

COMENT — This is a one-line description of the picture.

DEVICE — This array holds the natural device coordinates.

NDC — This array holds the normalised device coordinates.

WORLD — This array holds the world coordinates corresponding to the corners of the picture.

This information enables AGI to relate a particular position on the display surface to a particular position in world coordinates, and vice versa. The PNAME and COMENT fields are used to describe a 'picture', and the DEVICE, NDC, and WORLD fields are used to define its 'extent'.

Each time a picture is stored in the database for a particular workstation, it is added to the end of the PICTURE array. This array acts as a push-down stack; thus the 'last' picture is the one added most recently to the array, and this 'precedes' the one added before it. This order corresponds to the visual impression of new pictures obscuring older pictures on the display screen.

The database always has a 'current' picture and workstation associated with it. Each workstation also has a 'base' picture associated with it; this fills the display area and always exists. A new picture is always created inside the current picture. The current picture for each workstation is remembered between sessions.

AGI routines:

The 'objects' manipulated by the AGI library are 'pictures' in the database. The basic set of routines in the library begin with the prefix 'AGI_'. In the descriptions below, the routines will be identified by the characters which follow 'AGI_'. They fall naturally into the following groups:

Parameter routines —
These are the usual two routines used to integrate a subroutine library with the ADAM parameter system, i.e. **ASSOC** and **CANCL**. They correspond to the **OPEN** and **CLOSE** stand-alone routines. The 'picture' object is identified by a picture identifier 'picid'. The parameter with which the picture is associated is a graphics display device, normally called a workstation in GKS.

Activity routines —
These are the three routines that actually *do* something, rather than just navigate around the database.

SELP changes the identity of the current picture — most AGI routines simply use the current picture but don't change its identity.

NUPIC creates a new picture in the database and selects this as the new current picture. The information that must be provided is:

- The position of the new picture within the current picture.
- The 'NAME', 'COMENT', and 'EXTENT' of the new picture.

PDEL deletes all the pictures in the database for the current workstation.

Inquiry routines —
These nine routines obtain information about pictures. Their names (after the 'AGI_') all start with 'I'. One routine, **IBASE**, identifies the base picture for the current workstation. The other eight all obtain information about the current picture. Five of these, **ICURP**, **ILAB**, **INAME**, **ICOM**, and **IWOCO**, obtain the identifier, label, name, comment, and extent. Two others indicate whether or not the picture, **IPOBS**, or a set of test points, **ITOBS**, in the picture are obscured by a test picture. The last, **ISAMD**, says whether or not the picture is on the same workstation as a test picture.

Search routines —
These eight routines search the database for a picture with a given name, or which satisfies certain criteria. The routine names are of the form **RCxy**, where the 'x' specifies where to start the search and the direction of search, and 'y' can be used to specify that the picture must enclose a given test point.

Interfacing with other graphics components:

The basic routines do not assume the use of any particular graphics library. However, in order to make AGI easy to use it is desirable to integrate it with one or more graphics libraries. For instance, AGI and the graphics library need to be associated with the same workstation, and the entity addressed by the graphics library needs to be associated with the 'picture' addressed by AGI.

This integration has been done for SGS, by the addition of four routines which begin 'AGS_'. Two of them, **ACTIV** and **DEACT**, are used to initialise and close down SGS. The other two are **NZONE**, to create a new SGS zone corresponding to an AGI picture, and **SZONE**, to save the current SGS zone as a picture in the database.

This code will open AGI and create an SGS zone corresponding to the current picture:

```
CALL AGI_ASSOC('DEVICE', MODE, PICID, STATUS)
CALL AGS_ACTIV(STATUS)
CALL AGS_NZONE(ZONE, STATUS)
```

The associated interface file will contain specifications like these:

```
parameter  DEVICE
  ptype    'DEVICE'
  default  IKON
  prompt   'Name of display device'
  ...
endparameter
```

This will cause a workstation to be associated with parameter DEVICE, and PICID will point to the current 'picture' in the database entry for this workstation. SGS will be opened using the MODE to decide whether or not to erase the existing display, and an SGS zone will be opened which will correspond to the current picture in the database. You can then use PICID to address the database using AGI and AGS routines, and ZONE to address the display using SGS routines.

Similar routines exist to interface AGI with PGPLOT and IDI. These are prefixed by 'AGP_' and 'AGD_' respectively.

Full details of the AGI routines are given in SUN/48, and a comprehensive example of its use is given in SUN/95. You can examine the coding of other examples by looking at the DISPLAY and PHOTOM routines in the KAPPA source library KAPPA_DIR:KAPPA.TLB.

17.4 GNS — Graphics workstation name service

Almost any graphics program requires the user to identify which graphics device is to be used. When the GKS graphics package is used, graphics devices are identified by two integers: a 'workstation type' and a 'connection identifier'. No one can be expected to remember the workstation types of all available graphics devices, so a package called GNS has been provided that translates 'friendly' and easy to remember names into their GKS equivalents.

Most high-level graphics libraries, such as SGS, will perform the necessary name translation when a workstation is opened, so *unless you are writing programs that open GKS workstations directly (by calling* `GOPWK`*)* you will not need to call any GNS routines yourself.

The GNS library provides three sorts of service to the users and writers of graphics subroutine libraries and programs:

- Translating workstation names to their GKS equivalents.
- Generating a list of the names and types of graphics device available on a system.
- Answering inquiries about the properties of a particular graphics device; for example, its category (pen plotter, image display etc.) or its VMS device name.

More information on GNS and descriptions of its routines are given in SUN/57.

Demonstration program:

A complete list of all the workstation names defined on your system can be obtained by running the program `GNS_DIR:GNSRUN`. After listing all the GKS names along with a short description of each workstation, the program prompts for a workstation name. If one of the names in the list (or any other valid workstation name) is entered, a simple demonstration picture is plotted on the device selected. The IDI devices are then listed and again a demonstration plot can be generated by providing an appropriate device name.

If the program produces error messages you should report them to your Site Manager. However, you should not necessarily expect all workstations on the list to be usable. A device could, for example, be in use by someone else.

Chapter 18
Input/Output Systems

In ADAM programs, most data will be stored in HDS objects and handled by the Data system. However, it may occasionally be necessary to read and write files directly, a need which is met by the File system which comprises the FIO and RIO libraries. They are closely related and maintain a table of 'descriptors' of open files to enable automatic closedown. They also assist in the production of portable software.

The FIO routines handle sequential, formatted files to produce reports for subsequent listing. The RIO routines handle direct access, unformatted files. Each can open the other's files. When creating formatted files, it is possible to specify whether or not the file contains printer carraige control codes.

FIO and RIO use a common table of file descriptors, so RIO file descriptors may be used by appropriate FIO routines. For example, FIO_FNAME returns the filename associated with a file descriptor generated by either library.

A lot of astronomical data is stored on magnetic tape in formats which are different from the ADAM standard. Such tapes are called 'foreign tapes'. Some of these can be read and written by ADAM packages, such as FIGARO. However, a tape may exist in a format that cannot be read by an existing ADAM program, or a programmer may wish to write a program to process a foreign tape in his own way. To answer these needs, ADAM provides a subroutine library called MAG which enables a tape device to be controlled directly. The 'object' which is the parameter for this system is the 'tape drive'. It is addressed by a tape descriptor, TD.

18.1 FIO — Sequential file I/O

A file is opened by associating it with parameter PAR using the **FIO_ASSOC** routine:

```
    CALL FIO_ASSOC(PAR, MODE, FORM, RECSZ, FD, STATUS)
```

This has two more parameters (FORM and RECSZ) than most ASSOC routines. The mode of opening, MODE, may be 'READ', 'WRITE', 'UPDATE', or 'APPEND'. A new file is created if it is 'WRITE', or if it is 'APPEND' and the file does not exist. FORM specifies the type of formatting and the carriage control processing to be used when listing the file. RECSZ is used with the RECL keyword in the Fortran OPEN statement (used in the implementation) to define a maximum record length for the file; if RECSZ is zero, the RECL keyword is omitted from the OPEN statement. A file descriptor, FD, is used to specify the file in other FIO subroutines — it is not the same as a Fortran unit number. STATUS is the usual status value indicator.

Routine **FIO_CANCL** cancels the association between a parameter and a file.

Routines **FIO_READ** and **FIO_WRITE** read into, or write from, buffers containing character strings stored as formatted records. These buffers can be coded or decoded using the CHR library or Fortran internal I/O. If Fortran carriage control is specified for the file, the first character in the buffer should be set appropriately.

Routine **FIO_READF** reads a record from a file, but does not return the used length of the buffer. This makes reading much faster when the used length is not requried.

Example:

The following example shows how FIO routines are used to write a single formatted line to a file:

```
      SUBROUTINE EGFIO(STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INCLUDE 'FIO_PAR'
      INTEGER RECSZ
      CHARACTER*(FIO__SZMOD) MODE
      INTEGER FD
      INTEGER STATUS
*...........................................................................
* Use the 'WRITE' access mode
      MODE = 'WRITE'
* Create a file and open it for writing with default maximum record size
      CALL FIO_ASSOC('FILE', MODE, 'FORTRAN', 0, FD, STATUS)
* Write a record (note the initial carriage control space)
      CALL FIO_WRITE(FD, ' LINE TO BE WRITTEN', STATUS)
* Close the file and cancel the parameter
      CALL FIO_CANCL('FILE', STATUS)
* De-activate FIO
      CALL FIO_DEACT(STATUS)
      END
```

The corresponding interface file could contain the following specifications:

```
interface EGFIO
  parameter FILE
    type    'FILENAME'
    prompt  'Name of file to be created'
    ppath   'current,default'
    vpath   'prompt'
    default DATA_DIR:NEWFILE.DAT
  endparameter
endinterface
```

FIO_ASSOC obtains a value for parameter FILE and uses it as the filename in an internal call to FIO_OPEN. Checks on the validity of the filename are performed by the Fortran OPEN statement called internally.

Parameter FILE is defined as type FILENAME in the interface file and not as a character string, therefore the value given for 'default' does not need to be enclosed in quotes. Furthermore, the name of an HDS object containing the filename cannot be given.

When the program has finished with the file, it uses FIO_CANCL to close the file and cancel the parameter. If the file parameter is required again, to open the same file for reading perhaps, FIO_CLOSE should be used instead of FIO_CANCL.

The include file 'FIO_PAR' defines symbolic names for various constants which may be required by programs. For example, FIO__SZMOD is used in the above example to specify the length of the access mode string. Another useful symbolic name is FIO__SZFNAM (the maximum allowed length of a filename in FIO and RIO). If you want to test for explicit status values returned from FIO routines, include the statement:

```
INCLUDE 'FIO_ERR'
```

in the program. The status can then be tested by, for example:

```
IF (STATUS.EQ.FIO__ERROR) RETURN
```

Fortran I/O:

Fortran I/O will be required if unformatted records are to be stored in a sequential file, or formatted records in a direct access file. To do this, you will need to obtain the Fortran unit number associated with the file descriptor by using FIO_UNIT. Also, routine FIO_SERR can be used to convert a Fortran IOSTAT value to an appropriate FIO status value. For example, consider this program fragment:

```
* Open a direct access file in FORMATTED mode
      CALL RIO_ASSOC('TEMPFILE', 'APPEND', 'FORMATTED', 10, FDN, STATUS)
* If OK, get the Fortran unit number
      IF (STATUS.EQ.SAI__OK) THEN
        CALL FIO_UNIT(FDN, UNIT, STATUS)
* Write 5 formatted records - Must use Fortran I/O
        DO I = 1,5
          WRITE(UNIT, 1000, REC=I, IOSTAT=IOSTAT) I
1000      FORMAT(' RECORD', I2)
        END DO
* If OK, read the third record
        IF (IOSTAT.EQ.0) THEN
          READ(UNIT, 1001, REC=3, IOSTAT=IOSTAT) FORM, I
1001      FORMAT(A7,I2)
        END IF
* If the Fortran I/O failed, convert the error number
        IF (IOSTAT.NE.0) THEN
          CALL FIO_SERR(IOSTAT, STATUS)
* Else use the record
        ELSE
          ...
        END IF
      END IF
* Close the file and cancel the parameter
      CALL RIO_CANCL('TEMPFILE', STATUS)
* De-activate FIO
      CALL FIO_DEACT(STATUS)
```

If it is essential to use the Fortran OPEN or CLOSE statements, use FIO_GUNIT and FIO_PUNIT to obtain and release a logical unit number which does not clash with others. Note that there would be no FIO descriptor for such files.

18.2 RIO — Direct file I/O

The RIO system is very similar to the FIO system, and most of its routines have the same call sequence. For example:

```
CALL FIO_ASSOC(PAR, MODE, FORM, RECSZ, FD, STATUS)
CALL RIO_ASSOC(PAR, MODE, FORM, RECSZ, FD, STATUS)
```

There are some differences however. For example, in RIO_ASSOC, RECSZ specifies the fixed record length, while in FIO_ASSOC it specifies the maximum record length. Also, the read and write routines are different:

```
CALL FIO_READ(FD,        BUF,   NCHAR, STATUS)
CALL RIO_READ(FD, RECNO, NCHAR, BUF,   STATUS)

CALL FIO_WRITE(FD,               BUF, STATUS)
CALL RIO_WRITE(FD, RECNO, NCHAR, BUF, STATUS)
```

The RIO routines perform direct access I/O of unformatted byte arrays and need to specify a record number, RECNO. Also, they always need to specify the number of bytes, NCHAR, in a record.

Many FIO routines can be used on RIO files. For example,

```
    CALL FIO_FNAME(FD, FNAME, STATUS)
```

will get the file name, in FNAME, for any such file.

The last example in the previous section showed some RIO routines in use.

More detailed information on FIO and RIO can be found in SUN/143, which also contains routine specifications and error codes. A classified list of routines is given in Section 21.5.1.

18.3 MAG — Magnetic tape system

The routines in the MAG library have names which begin with the prefix 'MAG_', followed by characters which indicate their function. In this chapter, routine names will normally be written without the prefix. Thus, routine 'MAG_READ' will be referred to as 'READ'.

The functions provided by the MAG library can be classified as follows:

- **Device management** — These routines control the allocation of a tape drive to a program. Firstly, there are the usual two 'parameter' routines which integrate the library with the ADAM parameter system:

  ASSOC : Associate a parameter with a tape drive.
  CANCL : Cancel a parameter association.

  There is also the routine that is frequently used instead of CANCL:

  ANNUL : Annul a tape descriptor.

  Then there are two routines to control the allocation of a drive directly:

  ALOC : Allocate a tape drive.
  DEAL : De-allocate a tape drive.

  Finally, there are two routines to control the availability of a tape on a drive:

  MOUNT : Mount a tape.
  DISM : Dismount a tape

- **Tape positioning** — These routines control or establish a tape position. Firstly, there are three routines that position a tape relative to its current position:

  JUMP : Skip blocks.
  SKIP : Skip files.
  JEOV : Skip past an EOV marker.

  Then there are two routines that position a tape at an absolute position:

  REW : Position a tape at its beginning.
  MOVE : Position a tape at an absolute position.

  Finally, there are two routines to inquire about or set an absolute position:

> **POS** : Obtain an absolute position.
> **SET** : Set an absolute position.

- **I/O** — These routines transfer information between a program and a tape:

  > **READ** : Read a block.
  > **WRITE** : Write a block.
  > **WTM** : Write a tape mark.

The magnetic tape system uses the ADAM Error Strategy. If you want to test for specific MAG status values, include the statement:

```
    INCLUDE 'MAG_ERR'
```

in your program. A specific status can then be tested by, for example:

```
    IF (STATUS.EQ.MAG__EOV) RETURN
```

Linking:

To link a program with the MAG library, include ADAM_LIB:MAGLINK/OPT in the link command, for example:

```
    $ ALINK program,ADAM_LIB:MAGLINK/OPT
```

18.3.1 Device management

Device management is concerned with the allocation of a tape drive to a program, and the mounting of tapes on this drive.

Allocation and mounting are complicated by the fact that ADAM runs programs in sub-processes under the control of its own user interface. The problem is that a tape deck is allocated to a process, and when you have several processes active it matters which one the deck is allocated to. There are three ways of doing the allocation and mounting:

- At the DCL level, using the normal ALLOC and MOUNT commands.
- At the ICL level, using the ICL commands of the same name.
- Within the program, using the MAG routines ASSOC, ALOC, and MOUNT.

The first two levels have the advantage that generic device names can be used. However, they have the disadvantage that the initial tape position may not be available to the program. If absolute tape positions are not required, there is no problem. However, if they are required, special action must be taken to obtain them. One way is to use the REW routine to position the tape at its beginning. But the preferred way is to mount the tape by running the TAPEMOUNT program from the top-level process. For example, this can be done in DCL as follows:

```
    $ ADAMSTART
    $ ALLOC MT TAPE
    %DCL-I-ALLOC, _RLSTAR$MUC0: allocated
    $ TAPEMOUNT TAPE READ
    %MOUNT-I-MOUNTED,  mounted on _RLSTAR$MUC0:
    $ ICL
    ...
```

The first parameter of TAPEMOUNT is the name of the tape drive, and the second parameter is the access mode (READ or WRITE). When you have finished with the tape, you may dismount it using the TAPEDISM program:

```
$ TAPEDISM TAPE FALSE
```

The first parameter of TAPEDISM is the name of the tape drive, and the second parameter should be TRUE if tape is to be unloaded (the default), and FALSE otherwise.

18.3.2 Tape positioning

The information on a tape consists of a sequence of blocks. There is a special kind of block called a 'tape mark' which is used to indicate the end of a file. Two consecutive tape marks indicate 'end-of-volume' (EOV) which normally means that there is no further information on the tape.

The 'position' of a tape refers to the block which will be read next when the tape moves. This block will belong to a file, so an 'absolute position' can be specified by a file number (FILE) and a block number (BLOCK) within that file. When a tape is in the middle of being used it will have a 'current position', which is the absolute position it happens to be in at a particular moment. Every absolute position has a 'relative position' relative to the current position. Some MAG routines specify blocks by absolute position, and others by relative position. For example:

```
CALL MAG_MOVE(TD, 2, 'TRUE', 1, STATUS)
```

will position the tape at the first block of the second file and will assume it is being read in a forward direction, while:

```
CALL MAG_JUMP(TD, 5, STATUS)
```

will skip forward 5 blocks. A tape's absolute position can be found at any time by the call:

```
CALL MAG_POS(TD, FILE, START, BLOCK, STATUS)
```

If the position is unknown, FILE and BLOCK will be set to zero.

When reading forwards, the first data block in a file is specified by BLOCK=1. However, when reading backwards, the last data block in a file is specified by BLOCK=2.

Position memory:

The MAG package enables the position of a tape on a specific drive to be remembered between program runs. However, this can only be done if the tape is mounted using MAG_MOUNT. When a program has finished with a tape, it must call MAG_CANCL or MAG_ANNUL, at which time the tape must be in a known absolute position. MAG_MOUNT will initialize the position at FILE=1, START=.TRUE., BLOCK=1, while MAG_DISM will make the values undefined.

18.3.3 Example

The following example is a sketch of a program which reads a magnetic tape using MAG routines:

```
      SUBROUTINE EGMAG(STATUS)
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INCLUDE 'MAG_PAR'
      INTEGER TD
      INTEGER STATUS
*.........................................................
* Associate the TAPE parameter with an actual tape drive
      CALL MAG_ASSOC('TAPE', 'READ', TD, STATUS)
* Rewind the tape
      CALL MAG_REW(TD, STATUS)
* Process the tape
      ...
* Release the tape drive
      CALL MAG_ANNUL(TD, STATUS)
      END
```

The corresponding interface file could contain:

```
interface EGMAG
  parameter TAPE
    type    'TAPE'
    ptype   'DEVICE'
    prompt  'Tape Deck'
    ppath   'current,default'
    vpath   'prompt'
    default MTA0
  endparameter
endinterface
```

MAG_ASSOC requests that the associated tape drive be accessed in 'READ' mode, and sets a tape descriptor, TD, which is used by other MAG routines to identify the drive to be used. When the program has finished with the tape, it uses MAG_ANNUL to release the drive and store the tape position in the environment.

The error symbols and full routine specifications for the MAG library can be found in APN/1. Also, a classified list of the routines is given in Section 21.5.2.

Part V

REFERENCE

Chapter 19
ICL

19.1 Syntax

This section gives a summary of the ICL language syntax. Square brackets imply that the enclosed part of the construct is optional.

Operational Mode —

Direct:
Statements are executed as they are entered.

Procedure:
Statements are first entered into a procedure, then the procedure is executed.

Data types — **Real**

Integer

Logical

String

Expression— `value [operator value operator value ...]`

Value— **Constant:**

Real - `1.234E-5, 3.14159`
Integer - `123, %B100110, %O377, %Xffff`
Logical - `TRUE, FALSE`
String - `'This is a string', "So is this"`

Variable:

Name - Letter followed by letters, digits, or underscores (15 significant)

Function:
See below.

Operator—

1	`**`
2	`* /`
3	`+ -`
4	`= > < >= <= <> :`
5	`NOT AND OR &`

& performs string concatenation.
: is formatting operator (X:n:m is X converted to an n character string with m decimal places).

Statement— Normally one statement per line. A tilde (~) at the end of a line indicates that the statement continues on the next line.

Direct Statement:
These statements can be used in any operational mode.

Comment -
Any line starting with the characters '{' or ';'.

Immediate Statement -
= expression *(prints value of expression)*

Assignment Statement -
variable = expression *(assignment to variable)*

Command -
command_name p1 p2 . . . *(command or procedure call)*
In commands or procedure calls, parameters are separated by a comma or spaces. Expressions or variables should be enclosed in parentheses.

Control Statement:

These statements can only be used in the Procedure operational mode.

IF Statement -

```
IF expression
  statements
[ELSE IF expression
  statements]
[ELSE
  statements]
END IF
```

LOOP Statement -

```
LOOP
  statements
END LOOP

[LOOP] FOR var = exp1 TO exp2 [STEP exp3]
  statements
END LOOP

[LOOP] WHILE expression
  statements
END LOOP
```

You can escape from a loop via a BREAK statement.

PROCEDURE Statement -

```
[HIDDEN] PROC name p1 p2 ....
  statements
END PROC
```

EXCEPTION Statement -

```
EXCEPTION name
  statements
END EXCEPTION
```

Command Search Path— (1) User defined commands (defined by DEFSTRING, DEFUSER, DEFPROC)

(2) User written procedures (defined by PROC)

(3) ICL commands to control ADAM tasks (ALOAD, etc)

(4) ADAM commands (defined by DEFINE)

(5) Other ICL built-in commands (see below)

19.2 Commands

The commands which are likely to be of most use in data analysis are classified below. Commands relevant mainly to data acquisition are omitted. A full list is contained in SG/5.

Information & Escape— **HELP:** Display on-line documentation.

EXIT: Exit from ICL.

Defining user commands— **DEFSTRING:** Associate a command with an equivalence string.

DEFINE: Define a command to run an ADAM task.

DEFUSER: Associate a command with a user written subroutine.

DEFPROC: Associate a command with a procedure.

I/O— **Terminal I/O:**

PRINT: Output to the terminal.

INPUT: Input a string.

INPUTI: Input integers.

INPUTL: Input logical values.

INPUTR: Input real numbers.

Screen mode:

SET SCREEN: Set screen I/O mode.

SET NOSCREEN: Set normal I/O mode.

SET ATTRIBUTES: Set attributes for text written with the LOCATE command.

LOCATE: Write to screen at specified position.

CLEAR: Clear lines in a specified range.

Keyboard facilities:

KEY: Define an equivalence string for a key.

KEYTRAP: Specify trapping of a key.

KEYOFF: Turn off trapping of a key.

File I/O:

CREATE: Create a file and open it for output.

OPEN: Open an existing file for input.

APPEND: Open an existing file for appending.

CLOSE: Close a file opened with CREATE, OPEN, or APPEND.

WRITE: Write to a file.

READ: Read a string from a file.
READI: Read integers from a file.
READL: Read logical values from a file.
READR: Read real numbers from a file.

Access to DCL— General:

$: Issue a DCL command.
SPAWN: Spawn a subprocess to issue one or more DCL commands.
DEFAULT: Set/Show default directory.

Managing tape drives:

ALLOC: Allocate a device.
DEALLOC: Deallocate a device.
MOUNT: Mount a tape.
DISMOUNT: Dismount a tape.

Parameters— **SETPAR:** Set the value of a parameter where the program has an associated command.
GETPAR: Get the value of a parameter where the program has an associated command.
CREATEGLOBAL: Create a global parameter.
SETGLOBAL: Set the value of a global parameter.
GETGLOBAL: Get the value of a global parameter into an ICL variable.

Procedures— Listing and editing:

LIST: List a procedure.
PROCS: List procedure names.
VARS: List procedure variables.
EDIT: Edit a procedure.
SET EDITOR: Change the editor (TPU, EDT, or LSE) used by EDIT.

Saving and loading:

SAVE: Save a procedure.
LOAD: Accept commands from a saved procedure.
DELETE: Delete a procedure.

Tracing execution:

SET TRACE: Switch tracing of procedures ON.
SET NOTRACE: Switch tracing of procedures OFF.

Errors and Exceptions— **SIGNAL:** Signal an ICL exception.

Help system— **DEFHELP:** Define the source of help information.

Miscellaneous— **TASKS:** List all loaded tasks.
SAVEINPUT: Save previous input lines in a text file.
SETPRECISION: Set the number of decimal digits precision for unformatted conversions of real values to strings.

19.3 Functions

The following functions are available for use in expressions (N.B. The '$*$' symbol in the *Type* column stands for 'Real or Integer').

Mathematical Functions—

Name	*Type*	*Definition*
SIN(X)	Real	Trig. functions (x in radians).
COS(X)	Real	
TAN(X)	Real	
SIND(X)	Real	Trig. functions (x in degrees).
COSD(X)	Real	
TAND(X)	Real	
ASIN(X)	Real	$\sin^{-1} x \; where -1 \leq x \leq 1; -\pi/2 \leq result \leq \pi/2$.
ACOS(X)	Real	$\cos^{-1} x \; where -1 \leq x \leq 1; 0 \leq result \leq \pi$.
ATAN(X)	Real	$\tan^{-1} x; -\pi/2 \leq result \leq \pi/2$.
ATAN2(X1,X2)	Real	$\tan^{-1}(x_1/x_2); -\pi < result \leq \pi$.
ASIND(X)	Real	Inverse trig functions in degrees.
ACOSD(X)	Real	
ATAND(X)	Real	
ATAN2D(X1/X2)	Real	
SINH(X)	Real	Hyperbolic functions.
COSH(X)	Real	
TANH(X)	Real	
ABS(X)	*	$\mid x \mid$.
DIM(X1,X2)	*	Positive difference between x_1 and x_2.
EXP(X)	Real	e^x.
LOG(X)	Real	$\ln x$.
LOG10(X)	Real	$\log_{10} x$.
MIN(X1,X2,...)	*	Minimum of two or more arguments.
MAX(X1,X2,...)	*	Maximum of two or more arguments.
MOD(X1,X2)	*	Remainder when x_1 is divided by x_2.
SIGN(X1,X2)	*	Transfer of sign $\mid x_1 \mid$ Sign x_2.
SQRT(X)	Real	$\sqrt{x}$.

BIN(I,n,m)	String	Integer I formatted in binary into an n char strin with m significant digits.
DEC(I,n,m)	String	Integer I formatted in decimal into an n char stri with m significant digits.
HEX(I,n.m)	String	Integer I formatted in hexadecimal into an n cha with m significant digits.
OCT(I,n,m)	String	Integer I formatted in octal into an n char string with m significant digits.
DECL(S)	Real	Declination in radians from string in degrees, mi
DEC2S(R,NDP,SEP)	String	String in DMS with separator SEP and NDP deci on seconds, from Dec in radians.
RA(S)	Real	Right Ascension in radians from a string in hour
RA2S(R,NDP,SEP)	String	String in HMS with separator SEP and NDP deci on seconds, from RA in radians.

Formatting and String Handling Functions—

CHAR(I)	String	Character with ASCII value I.
ICHAR(S)	Integer	ASCII value of first character of string S.
INDEX(S1,S2)	Integer	Position of first occurrence of string S2 in string S
LEN(S)	Integer	Length of string.
ELEMENT(I,DELIM,S)	String	Ith element of delimited string.
SUBSTR(S,n,m)	String	Substring of S beginning at n of length m.
SNAME(S,n,m)	String	Name derived by concatenating string S with int formatted into m characters including leading ze
UPCASE(S)	String	String S converted to upper case.
LGE(S1,S2)	Logical	True if S1 $\geq$ S2 (ASCII collating sequence).
LGT(S1,S2)	Logical	True if S1 $>$ S2 (ASCII collating sequence).
LLE(S1,S2)	Logical	True if S1 $\leq$ S2 (ASCII collating sequence).
LLT(S1,S2)	Logical	True if S1 $<$ S2 (ASCII collating sequence).

Bitwise Logical Operations—

IAND(I1,I2)	Integer	Bitwise AND.
IOR(I1,I2)	Integer	Bitwise OR.
IEOR(I1,I2)	Integer	Bitwise Exclusive OR.
INOT(I)	Integer	Bitwise Complement.

Type Conversion and Inquiry Functions—

FLOAT(I)	Real	Integer I converted to real.
IFIX(X)	Integer	X converted to integer by truncation.
INT(X)	Integer	X converted to integer by truncation.
NINT(X)	Integer	Nearest integer to X.
INTEGER(X)	Integer	X converted to integer.
LOGICAL(X)	Logical	X converted to logical.
REAL(X)	Real	X converted to real.
STRING(X)	String	X converted to string.
TYPE(X)	String	The type of X ('REAL', 'INTEGER', 'LOGICAL', 'STRING', or 'UNDEFINED').
UNDEFINED(X)	Logical	TRUE if X is undefined.

Miscellaneous Functions—

DATE()	String	Current date.
TIME()	String	Current time.
GETNBS(S)	Any	Value of noticeboard item.
GET_SYMBOL(S)	String	Value of DCL symbol.
INKEY()	Integer	Key value of last key trapped (Screen mode only).
KEYVALS(S)	Integer	Value of key with name S.
OK(stat)	Logical	True if VMS Status OK.
FILE_EXISTS(S)	Logical	True if file S exists.
RANDOM(I)	Real	Random number between 0 and 1 from seed variable I.
VARIABLE(proc,X)	Any	Returns value of variable X of procedure proc.

19.4 Exceptions

Name	*Description*
ADAMERR	An error has occurred in an ADAM task. An associated message will be output.
ASSNOTVAR	An assignment has been made to a procedure formal parameter which does not correspond to a variable in the procedure call.
CLOSEERR	Error closing text file.
CONVERR	Error converting RA or Dec to string or vice versa.
CTRLC	A Control-C has been entered on the terminal.
DEVERR	Error allocating or mounting device.
EDITERR	Attempt to use the LSE or TPU editors when they are not available.
EOF	End of file encountered on text file operation.
FIGERR	Error in a FIGARO program, or FIGARO not available.
FLTDIV	Floating point division by zero.
FLTOVF	Floating point overflow.
IFERR	The expression in an IF or ELSE IF statement does not evaluate to a logical value.
INTOVF	Integer overflow.
INVARGMAT	Invalid argument to a mathematical function.
INVSET	Invalid SET command.
LOGZERNEG	Logarithm of zero or negative number.
NBSERR	Error in GETNBS or PUTNBS.
OPENERR	Error opening text file.
OPNOTLOG	Operands of a logical operator (AND, OR *etc.*) are not logical values.
OPNOTNUM	Operands of a formatting operation (:) are not numeric.
PROCERR	Unrecognized procedure or command name.
READERR	Error reading from text file.
RECCALL	Attempt to make a recursive call of a procedure.
SCREENERR	Error in screen mode I/O.
SQUROONEG	Square root of negative number.
STKOVFLOW	ICL's stack has overflowed.
STKUNDFLOW	ICL stack underflow — if this occurs it indicates an internal error in ICL.
TOOFEWPARS	Not enough parameters for a function or command.
TOOMANYPARS	Too many parameters for a procedure or command.
UNDEFVAR	Attempt to use an undefined variable — *i.e.* one that has not yet had a value assigned.
UNDEXP	Undefined Exponentiation.
USERERR	Error accessing a routine defined using DEFUSER, or error during such a routine.
WHILEERR	The expression in a WHILE statement does not evaluate to a logical value.
WRITERR	Error writing to text file.

Chapter 20 Applications

This chapter lists the commands in the application packages mentioned in Chapter 4.

20.1 ASTERIX — X-ray data analysis [SUN/98]

Shortform and macro commands are shown with suffixes of a cross and an asterisk respectively.

Instrument Interfaces — Exosat:

- **BCKBOXEXOLE*:** Draw a source and background box on an Exosat LE image and sort the data within these regions.
- **BOXEXOLE*:** Draw a box on an Exosat LE image and sort the data within this region.
- **EXOLESORT:** Sort Exosat LE raw data files into Asterix datasets.
- **EXOMESORT:** Sort Exosat ME raw data files into Asterix datasets.
- **EXORESP:** Write an EXOSAT ME response matrix into a dataset.
- **EXOSUBPH:** Subtract background for ME data.
- **READEXO:** Read Exosat FOT's onto disc.
- **SCANEXO:** Scan an Exosat tape and produce a summary.
- **SHOWEXO:** Produce a summary of observations on a disc directory.
- **TIMEXOLE*:** Plot a time series, select a time window, and sort LE data between these values.
- **TIMEXOME*:** Same for the ME data.

Rosat WFC:

- **S2ADDOWNER:** Add an ownership flag to an S2 catalogue using WFC sky division.
- **S2ADDPFLAG:** Add a processing flag to an S2 catalogue.
- **S2CATLIST:** List contents of an S2 format catalogue.
- **S2CATMERGE:** Merge two S2 format catalogues to create a third.
- **S2FILTMERGE:** Merge source search results from each filter into an S2 catalogue.
- **S2GIDECODE:** Unpack cross-reference data from an S2 catalogue into SCAR global index.
- **S2GIENCODE:** Pack a SCAR global index into an S2 catalogue.
- **WFCBACK:** Produce a background subtracted image given raw image.
- **WFCSPEC:** Produce a WFC spectral file.

Rosat XRT:

- **PREPXRT:** Reformat the output from XRTDISK .
- **XRTCORR:** Correct XRT files.
- **XRTDISK:** Read XRT FITS disk files.
- **XRTORB:** Reformat an XRT orbit file.
- **XRTRESP:** Write XRT response into file.
- **XRTSUB:** Subtract XRT files.
- **XSORT:** Sort raw XRT data.

Event Processing — **EVBIN:** Create a binned dataset from an event dataset.

- **EVCSUBSET:** Extract a circular or annular region from an event dataset.
- **EVLIST:** Display the DATA_ARRAY component of all lists in an event dataset.
- **EVMERGE:** Merge two or more event datasets.

EVPOLAR: Bin an event dataset into a polar binned dataset.
EVSIM: Generate a simulated event dataset.
EVSUBSET: Extract linearly a subset from an event dataset.

Binned Dataset — **AXFLIP:** Reverse directions of any or all of dataset's axes.
AXORDER: Re-order axes of a dataset.
AXSHOW: Display axes.
BINMERGE: Merge up to ten datasets.
BINSUBSET: Subset a binned dataset.
ENMAP: Produce an energy map for plotting.
INTERP: Reconstitute bad pixels by spline interpolation.
MEANDAT: Find the mean of several datafiles.
POLYFIT: Fit 1-d polynomial(s) to a dataset.
PROJECT: Project data along one axis.
RATIO: Give the ratio of two bands on any axis in an n-d array.
REBIN: Rebin a binned dataset.
SCATTERGRAM: Produce scatter plot of one array versus another.
SETRANGES: Set ranges to be used by a subsequent program.
SIGNIF: Change an input dataset to its weighted significance.
SMOOTH: Smooth an n-d datafile with a user-selectable mask.
SYSERR: Add a constant percentage to the variance of each point.
VALIDATE: Basic validation of binned dataset.

Conversion — **ASTCONV:** Convert between old & new Asterix binned datasets.
AST2XSP: Convert Asterix spectral files into XSPEC format.
ASTQDP: Allow an Asterix file to be used within QDP.
EVBIN: Create a binned dataset from an event dataset.
EXPORT: Output one or more datasets to ASCII file.
IMPORT: Read an ASCII text file into an Asterix binned dataset.
TEXT2HDS: Convert columns in a text file into HDS arrays.

Display — **BINLIST:** Display a 1-d binned dataset.
EVLIST: Display the DATA_ARRAY component of all lists in an event dataset.
HEADER: Display header & processing information in a dataset.
HISTORY: Display history of a dataset.

Maths — **ARITHMETIC:** Perform basic arithmetic (+,−,/,*) on two data objects.
ADD+: Invoke ARITHMETIC in + mode.
SUBTRACT+: Invoke ARITHMETIC in − mode.
MULTIPLY+: Invoke ARITHMETIC in * mode.
DIVIDE+: Invoke ARITHMETIC in / mode.
OPERATE: Perform operations e.g. Log_{10} on a dataset.

Time Series Analysis — **ACF:** Auto-correlation program.
BARYCORR: Barycentric correction.
CROSSCOR: Cross-correlate two 1-d series.
CROSSPEC: Compute the cross spectrum of two 1-d datasets.
DIFDAT: Difference adjacent data points in an array.
DYNAMICAL: Find power spectrum of successive segments of a time series.
FOLDAOV: Period search ANOVA folding.
FOLDBIN: Fold a time-series into phase bins at a given period.
FOLDLOTS: Epoch folding, period search algorithm.
LOMBSCAR: Lomb-Scargle power spectral analysis.
PHASE: Convert a time-series into a phase series.
POWER: Find the power spectrum of a full unweighted 1-d dataset.

SINFIT: Find a periodogram of irregularly spaced 1-d data.
STREAMLINE: Strip bad quality data out of a file.
TIMSIM: Simulate a time series.
VARTEST: Test a time series for variability.

Image Processing — Interactive:

IAZIMUTH: Produce azimuthal distribution.
IBLUR: Smooth an image.
IBLURB+: Smooth with a box filter.
IBLURG+: Smooth with a Gaussian.
IBOX: Define rectangular section of image.
IBOXSTATS*: Select region and give statistics.
ICENTROID: Find the centroid in a given region.
ICIRCSTATS*: Get flux from a circular region.
ICLEAR: Clear current plotting zone.
ICLOSE: Close down image processing system.
ICOLOUR: Change colour table.
ICONTOUR: Contour current image.
ICURRENT: Display the current position, image etc.
IDISPLAY: Display current image.
IGRID: Put grid over image in specified coords.
IHARD: Hard copy of current plot.
IHIST: Histogram pixels inside current box.
ILIMITS: Change axis limits on 1-d plot.
IMARK: Mark sources on image.
INEW: Load new image.
INOISE: Add Gaussian noise to image.
IPATCH: Patch bad quality pixels in image.
IPEAKS: Find peaks in image.
IPEEK: Take a peek at selected bit of image.
IPEEKQ+: Look at quality.
IPEEKV+: Look at variances.
IPGDEF: Change basic graphics properties.
IPLOT: Display current 1-d plot.
IPOSIT: Set the current position.
IPREVIOUS: Go back to previous image.
IPSF: Put PSF profile on 1-d plot.
IRADIAL: Produce radial plot.
IREDISPLAY+: Redisplay current image.
IREMOVE: Remove sources from image.
IREPLOT+: Redisplay current 1-d plot.
ISAVE: Save current image to file.
ISAVE1D: Save current 1-d data to file.
ISCALE: Rescale image.
ISEP: Calculate separation between two positions.
ISLICE: Take 1-d slice from image.
ISTART: Start up image processing system.
ISTATS: Give basic statistics on pre-selected region of image.
ISTYLE: Set plotting style of 1-d plot.
ISURFACE: Display image as 3-d surface.
ITITLE: Change title of current image or plot.

IUNZOOM*: Redisplay whole image.
IWHOLE: Select whole image.
IZAP: Remove individual pixels from image.
IZONES: Change zones on display surface.
IZOOM*: Zoom in on section of image.

Non-interactive:

IDMSTOD*: Convert dd:mm:ss to decimal degrees.
IDTODMS*: Convert decimal degrees to dd:mm:ss.
IDTOHMS*: Convert decimal degrees to hh:mm:ss.
IHMSTOD*: Convert hh:mm:ss to decimal degrees.
IMOSAIC: Merge several non-congruent images.
IPOLAR: Produce polar surface brightness profile of image.

Parameter — **GLOBAL:** Display current values of all global parameters.

Spectral analysis — **FREEZE:** Freeze parameter(s) in spectral model.
IGNORE+: Allow spectral channels to be excluded from fitting.
RESTORE+: Reinstate spectral channels for fitting.
SDATA: Define datasets to be fitted.
SERROR: Evaluate confidence region for spectral model parameters.
SFIT: Fit spectral model to one or more datasets.
SFLUX: Evaluate flux from defined model over energy band.
SMODEL: Allow user definition of multi-component spectral model.
SPLOT: Plot data, fits and residuals.
THAW: Free spectral model parameter (after FREEZEing).

Statistical analysis — **BINSUM:** Integrate dataset – use for cumulative distributions.
COMPARE: Compare two datafiles or a file and a model.
FREQUENCY: Produce histogram of values in data array.
KSTAT: Calculate Kendall K statistic for 1-d dataset.
STATISTIX: Find mean, standard deviation etc. of data array.

Quality processing — **CIGNORE:** Set quality bad in circular region.
CQUALITY: Perform quality manipulation on circular region.
CRESTORE: Set quality good in circular region.
IGNORE+: Invoke QUALITY in IGNORE mode; sets temporary bad quality bit.
MAGIC: Set magic values.
QUALITY: General quality manipulation application.
RESTORE+: Invoke QUALITY in RESTORE mode; clears temporary bad quality bit.
SETQUAL+: Invoke QUALITY in SET mode; sets quality to specified value.

HDS editor — **HCOPY:** Copy an HDS object recursively.
HCREATE: Create an HDS object in a file.
HDELETE: Delete an HDS object.
HDIR: Display the components of an HDS file or structure.
HDISPLAY: Display the contents of an HDS primitive.
HFILL: Fill a primitive object with one value.
HMODIFY: Change the value(s) in an existing object.
HREAD: Read from an ASCII (or binary) file to an HDS object.
HRENAME: Rename an HDS object.
HRESET: Set values in an HDS primitive to 'undefined'.
HRESHAPE: Alter size of dimensions in an HDS primitive array.
HRETYPE: Change type of a structured object.
HSE: Interactive HDS screen editor – see HSEHELP.
HTAB: Simultaneously display several HDS primitive vectors.

HWRITE: Write an HDS primitive to an ASCII (or binary) file.

Source Searching — **BSUB:** Background subtraction program.

CREPSF: Create a PSF file.

PSS: Search a binned dataset for sources.

PSSPAR+: Invoke PSS in parameterise mode.

SSANOT+: Annotate an image given source search results.

SSCARIN: Export source search results to binary SCAR catalogue.

SSDUMP: Dump source search results to an ASCII file.

SSMERGE: Merge two or more source results together.

SSZAP: Set quality bad around sources found by PSS.

XPSSCORR: Exposure-correct XRT source fluxes produced by PSS.

GRAFIX display — **DEVICES:** Show the available graphics display devices.

DOPEN+: Open device (variant of DEVICES).

DCLOSE+: Close device (variant of DEVICES).

DRAW: Display specified (or current) dataset.

QDRAW+: Draw quick and simple plot (variant of DRAW).

RDRAW+: Plot raw data only (variant of DRAW).

MULTI: Create multi-graph dataset from binned datasets.

AMULTI+: Add another binned dataset to a multi-graph dataset.

DMULTI+: Remove a dataset from a multi-graph dataset (variants of MULTI).

DESIGN: Display specified plots within a multi-graph dataset.

LAYOUT: Specify number of plots in x & y in multi-graph dataset.

POSIT: Specify position of plot in NDC or cm.

XLOG+: Set x-axis logarithmic (variant of AXES).

YLOG+: Set y-axis logarithmic (variant of AXES).

XYLOG+: Set both axes logarithmic (variant of AXES).

AXES: Set plotting attributes for axes.

XAXIS+: Set attributes for x-axis (variant of AXES).

YAXIS+: Set attributes for y-axis (variant of AXES).

LABEL: Set labels on plot.

XLABEL+: Set x-label (variant of LABELS).

YLABEL+: Set y-label (variant of LABELS).

LEGEND: Add, delete or modify legend lines.

ALEGEND+: Add a legend line (variant of LEGEND).

DLEGEND+: Delete a legend line (variant of LEGEND).

ANOTATE: Place text in graph (binned dataset) at specified position.

POLYLINE: Draw 1-d image as polyline.

MARKER: Draw 1-d dataset with specified PGPLOT marker.

STEPLINE: Draw 1-d dataset as histogram.

ERRORS: Draw error boxes of specified shape on 1-d dataset.

PIXEL: Draw 2-d image as pixels.

CONTOUR: Draw 2-d dataset as contours.

THREED: Draw 2-d dataset as quasi 3-d plot.

COLTAB: Manipulate colour table for 2-d plot.

COLBAR+: Cause colour bar to be put on 2-d plot.
GREYSCALE+: Set colour table to greyscale.
SKYGRID: Put coordinate grid on 2-d plot.

CURSOR: Put interactive cursor on screen.
SHAPES: Put a shape onto a displayed image.
ZOOM*: Set plot limits interactively.

PGDEF: Set global PGPLOT default attributes.

20.2 CCDPACK — CCD data reduction [SUN/139]

CALCOR: Perform dark or flash count corrections for a list of NDFs.

CCDBATCH: Prepare a CCDPACK command procedure for submission to batch.

CCDCLEAR: Clear the current globals parameters.

CCDNOTE: Add a note to the log file.

CCDSETUP: Set up the CCDPACK global parameters.

CCDSHOW: Display the current values of the CCDPACK global parameters.

DEBIAS: Debiass lists of NDFs either by bias NDF subtraction or by interpolation; apply bad data masks; extract a subset of the data area; produce variances; apply saturation values.

FLATCOR: Perform the flatfield correction on a list of NDFs.

LISTLOG: List the contents of a logfile.

MAKEBIAS: Produce a bias calibration NDF.

MAKECAL: Produce calibration NDFs for flash or dark counts.

MAKEFLAT: Produce a flatfield NDF.

20.3 CONVERT — Data format conversion [SUN/55]

DIPSO2NDF: DIPSO to NDF format conversion.

NDF2DIPSO: NDF to DIPSO format conversion.

DST2NDF: Figaro version 2 to NDF format conversion.

NDF2DST: NDF to Figaro version 2 format conversion.

BDF2NDF: Interim (BDF) to NDF format conversion.

NDF2BDF: NDF to Interim (BDF) format conversion.

20.4 DAOPHOT — Stellar photometry [SUN/42]

This consists of the portable package itself, together with three additional routines to display results obtained by DAOPHOT on an image display.

DAOPHOT: this responds to the following commands :

- **ADDSTAR:** Add synthetic stars to an image.
- **ALLSTAR:** An alternative to the iterative profile fitting routine NSTAR.
- **APPEND:** Concatenate two data files.
- **ATTACH:** Specify the disk file name of the required picture.
- **DUMP:** Display raw values from part of your picture on your terminal.
- **EXIT:** Leave the program.
- **FIND:** Find all the stars above a certain threshold.
- **FUDGE:** Change image data to fix minor problems such as cosmic ray hits (last resort).
- **GROUP:** Group stars together, on the principle that if two stars are close enough that the light from one will influence the profile-fit of the other then they should be in the same group.
- **HELP:** Produce a simple list of the commands which are available.
- **LIST:** Display the image header of 'Caltech data-structure files'.
- **MONITOR:** Restore reporting of progress to your terminal.
- **NOMONITOR:** Switch off the reporting of progress to your terminal.
- **NSTAR:** Perform multiple, simultaneous profile-fitting photometry.
- **OFFSET:** Add fixed values to every X and Y co-ordinate in a data file.
- **OPTIONS:** Set values for various parameters to ones suitable for the given data.
- **PEAK:** Perform profile-fitting for a single star.
- **PHOTOMETRY:** Perform concentric aperture photometry in the specified image.
- **PICK:** Select a set of reasonable candidates for PSF stars.
- **PSF:** Obtain a point-spread function from the given image.
- **SELECT:** Cut down the size of some of the groups found by GROUP (if there are too many stars in a group).
- **SKY:** Estimate the sky brightness in the image.
- **SORT:** Re-order the stars in one of the lists produced by DAOPHOT according to various criteria.
- **SUBSTAR:** Remove suitable multiples of a PSF, at specified locations, from an image.

DAOGREY: Display the image data on a suitable device.

DAOPLOT: Take a file produced by DAOPHOT and overlay the positions of objects in the file on top of the grey-scale image produced by DAOGREY.

DAOCURS: Display a cursor over the grey image produced by DAOGREY so that positions can be read off the screen.

20.5 FIGARO — General spectral reduction [SUN/86]

I/O — Input: ALASIN : Read a spectrum in ALAS (Abs. Line Analysis System) format.
FIND : Read an image in IPCS format.
FITS : Read data from a FITS format tape.
FITSLIST : List the FITS keywords in a data file.
ICOR16 : Correct 16-bit data from signed to unsigned range.
R4S : Read an image in 4-Shooter format.
RBAZ : Read a DBSP CCD tape (in BAZ's format).
RCSHEC : Read raw Shectograph data in Copyshec disk format.
RCTIO : Read images and spectra written in CTIO format.
RDFITS : Read a file in AAO de facto 'Disk FITS' format.
RDIPSO : Read a file in DIPSO/IUEDR/SPECTRUM format.
REWIND : Rewind either the input or output tape.
RJAB : Read an image in JAB's image format.
RJKM : Read a spectrum in JKM's floppy disk format.
RJPL : Read an image from a JPL linear array camera tape.
RJT : Read an image in JT's 'Disk FITS' format.
RLOL : Read a spectrum from disk in Lolita output format.
RNTYB : Read images written around 19Feb84 in 'almost' TYB format.
RPDM : Read an image in FORTH Picture Disk Manager format.
RPFUEI : Read an image from a Pfuei format tape.
RSHEC : Read a raw Schectograph data tape.
RSIT : Read 60" SIT data from tape (images or spectra).
RSPDM : Read a spectrum in FORTH PDM/TYB format.
RSPICA : Read a spectrum or image from a Spica Memory file.
RTYB : Read an image in TYB format.
RXMIC : Read an image in X-ray microscope format.
SKIP : Skip files on tape.
SFIND : Read data from a tape in SDRSYS format.
STARIN : Read an image or spectrum from a Starlink BDF file.
TABLE : List contents of a Spica memory file.
TAPE : Set the tape drive to be used for input.

Output: ALASOUT : Write a spectrum in ALAS (Abs. Line Analysis System) format.
STAROUT : Write an image or spectrum in Starlink BDF file format.
TAPEO : Set the tape drive to be used for output.
WAIS : Write a spectrum in AIS's PLAY format.
WDFITS : Write an image in the AAO de facto 'Disk FITS' format.
WDIPSO : Write an image in DIPSO/IUEDR/SPECTRUM format.
WIFITS : Write an image (or spectrum) to tape in FITS format.
WJAB : Write an image in JAB's Pamela format.
WJT : Write an image in JT's 'Disk FITS' format.
WLOL : Convert a spectrum to Lolita format.
WPDM : Write an image in FORTH picture disk manager format.
WSPICA : Write an image or spectrum into a Spica Memory file.

DISPLAY — On graphics/image devices: ARGS : Select which ARGS device is to be used.
BLINK : Blink a displayed image pair.
BPLOT : Plot a 'build' file generated by SPLOT.
CCUR : After SPLOT, use graphics cursor to indicate data values.

COLOUR : Set image display colour table.
CPAIR : Colour a displayed image pair.
CPOS : Select points with image display cursor.
DVDPLOT : Plot the data in one file against the data in another.
ELSPLOT : Produce a long error bar plot of a spectrum.
ESPLOT : Produce an error bar plot of a spectrum.
HARD : Set the file name for hard copy output.
HOPT : Histogram-optimization of an image.
ICONT : Produce a contour map of an image.
ICUR : Use ARGS cursor to show x,y and data values.
IGREY : Produce a grey-scale plot of an image.
IKON : Specify the IKON device to use as an image display.
IMAGE : Display an image on the selected image display.
IMAGE2 : Display a pair of separate images on the image display.
IMAGEPS : Create a Postscript file giving a grey scale image picture.
IMPAIR : Display an image pair on the selected image display.
IPLOTS : Plot successive cross-sections of an image, several to a page.
ISPLOT : Plot successive cross-sections through an image.
LSPLOT : Plot hardcopy spectrum of specified size (up to 3 metres).
MSPLOT : Plot a long spectrum as a series of separate plots.
PAIR : Combine two images to form an image pair.
SOFT : Set the device/type for terminal graphics.
SPLOT : Plot a spectrum.
VAPLOT : Produce an ARGS plot of a spectrum with scales under trackerball control.
XCUR : Use cursor to delimit part of a spectrum.
ZOOM : Zoom and pan an image (in a way compatible with ICUR).

On non-graphics devices: **EXAM** : Examine the contents of a data object.
ILIST : List the data in an image (or spectrum).
ISTAT : Provide some statistics about an image (max, min etc.).

CALIBRATION — **Flat field calibration:** **CFIT** : Generate a spectrum using the cursor.
FF : Flat field an image (uses JT's algorithm).
FFCROSS : Cross-correlate an image and a flat field (mainly IPCS data).
MASK : Generate a mask spectrum, given a spectrum and a mask table.
MCFIT : Fit a continuum to a spectrum, given a mask spectrum.
ISXDIV : Divide a continuum into a spectrum, given a mask spectrum.

Wavelength calibration: **ARC** : Identify manual arc line interactively.
ECHARC : Fit an echelle arc.
EMLT : Fit Gaussians to the strongest lines in a spectrum.
FSCRUNCH : Rebin data with a disjoint wavelength coverage to a linear one.
IARC : Fit all spectra in a 2-d arc, given fit to single spectrum.
ISCRUNCH : Rebin an image to linear wavelength scale given IARC results.
ISCRUNI : Like ISCRUNCH, but interpolate between two IARC result sets.
IXSET : Create a 2-d X-array from an IARC fit.
LXSET : Set X-array of spectrum/image to specified range.
SCRUNCH : Rebin a spectrum to a linear wavelength range.
VACHEL : Convert wavelength for air to vacuum, and/or recession velocity.
XCOPI : Like XCOPY, but interpolate X-data from 2 files.
XCOPY : Copy X-info (e.g. wavelengths) into a spectrum.

Flux calibration: **ABCONV** : Convert a spectrum from Janskys into AB magnitudes.
CALDIV : Generate a calibration spectrum from continuum standard spectra.

CFIT : Generate a spectrum using the cursor.
CSET : Set regions of a spectrum to a constant value interactively.
CSPIKE : Create a calibration spiketrum, given spiketrum & standard spectrum.
FIGSFLUX : Flux calibrate a FIGS spectrum.
FLCONV : Convert a spectrum in Janskys into one in ergs/cm**2/s/Å.
FWCONV : General unit conversion for spectra.
GSPIKE : Generate a spiketrum from a table of values.
INTERP : Interpolate between the points of a spiketrum $\rightarrow$ a spectrum.
IRFLUX : Flux calibrate an IR spectrum using a black-body model.
LINTERP : Interpolate linearly between spiketrum points $\rightarrow$ spectrum.
NCSET : Set a region of a spectrum to a constant.
SFIT : Fit a polynomial to a spectrum.
SPFLUX : Apply a flux calibration spectrum to an observed spectrum.
SPIED : Edit a spiketrum interactively.
SPIFIT : Fit a global polynomial to a spiketrum $\rightarrow$ a spectrum.

Distortion measurement and correction: **CDIST** : Correct S-distortion using SDIST results.
ICUR : Show x,y and data values using ARGS cursor.
FINDSP : Locate fibre spectra in an image.
ODIST : Overlay an SDIST fit on another image.
OFFDIST : Apply an offset to an SDIST fit.
OVERPF : Overlay a FINDSP fit on another image.
POLEXT : Extract fibre spectra from an image after a FINDSP analysis.
SDIST : Analyse an image containing spectra for S-distortion.

Extinction coefficients: **EXTIN** : Correct a spectrum for atmospheric extinction.
GSPIKE : Generate a spiketrum from a table of values.
LINTERP : Interpolate linearly between spiketrum points $\rightarrow$ spectrum.

B star calibration: **BSMULT** : Remove atmospheric band using a B star calibration spectrum.
CFIT : Generate a spectrum using the cursor.
CSET : Set regions of a spectrum to a constant value interactively.
MASK : Generate a mask spectrum given a spectrum and a mask table.
MCFIT : Fit a continuum to a spectrum, given a mask spectrum.
NCSET : Set a region of a spectrum to a constant.

MANIPULATION — **Arithmetic:** **CLIP** : Clip data above and below a pair of threshold values.
CONTRACT : Reduce size of files by contracting arrays.
EXPAND : Expand files reduced by CONTRACT.
IADD : Add two images (or two spectra).
ICADD : Add a constant to an image.
ICDIV : Divide an image by a constant.
ICMULT : Multiply an image by a constant.
ICONV3 : Convolve an image with a 3x3 convolution kernel.
ICSUB : Subtract a constant from an image.
IDIFF : Take the 'differential' of an image.
IDIV : Divide two images (or two spectra).
ILOG : Take the logarithm of an image.
IMULT : Multiply two images (or two spectra).
IPOWER : Raise an image to a specified power.
IREVX : Reverse an image (or spectrum) in the X-direction.
IREVY : Reverse an image (or spectrum) in the Y-direction.
ISHIFT : Apply a linear x and a linear y shift to an image.
ISMOOTH : Smooth a 2-d image using 9-point smoothing algorithm.

ISTRETCH : Stretch and shift an image in X and Y.
ISUB : Subtract two images (or two spectra).
ISUBSET : Produce a subset of an image.
ISUPER : Produce a superset of an image.
ISXADD : Add a spectrum to each X direction x-sect of an image.
ISXDIV : Divide a spectrum into each X direction x-sect of an image.
ISXMUL : Multiply each X direction image x-sect by a spectrum.
ISXSUB : Subtract each X direction image x-sect from a spectrum.
ISYADD : Add a spectrum to each Y direction x-sect of an image.
ISYDIV : Divide a spectrum into each Y direction x-sect of an image.
ISYMUL : Multiply each Y direction image x-sect by a spectrum.
ISYSUB : Subtract each Y direction image x-sect from a spectrum.
IXSMOOTH : Smooth in X direction by Gaussian convolution.
ROTX : Rotate data along the X-axis.

Complicated: ADJOIN : Append two spectra (strictly a merge by wavelength value).
BCLEAN : Remove automatically bad lines & cosmic rays from CCD data.
CFIT : Generate a spectrum using the cursor.
CLEAN : Patch bad lines and bad pixels in an image interactively.
COADD : Form the spectrum which is the mean of the rows in an image.
COMBINE : Combine two spectra, adding with weights according to errors.
COSREJ : Reject cosmic rays from a set of supposedly identical spectra.
CROBJ : Create a data object or file.
CSCALE : Estimate scale factor for 2 images using ARGS.
FSCRUNCH : Rebin data with a disjoint wavelength coverage to a linear one.
HIST : Produce a histogram of data value distribution in an image.
HOPT : Histogram optimize an image.
ICONV3 : Convolve an image with a 3x3 convolution kernel.
ICOR16 : Correct 16-bit data from signed to unsigned range.
IDIFF : Take the 'differential' of an image.
IERASE : Subtract erase line from a CCD image.
IREVX : Reverse an image (or spectrum) in the X direction.
IREVY : Reverse an image (or spectrum) in the Y direction.
MEDFILT : Apply a median filter to an image.
MEDSKY : Take the median of a number of images.
PAIR : Combine two images to form an image pair.
POLYSKY : Fit and subtract sky from a long slit spectrum.
ROTATE : Rotate an image through 90 degrees.
SCNSKY : Calculate a sky spectrum for a scanned CCD image.
SCROSS : Cross-correlate two spectra & get relative shift.
SCRUNCH : Rebin a spectrum to a linear wavelength range.
SFIT : Fit a polynomial to a spectrum.
SURFIT : Fit an image using bi-cubic splines.

Complex data: BFFT : Take the reverse FFT of a complex data structure.
CMPLX2I : Extract the imaginary part of a complex data structure.
CMPLX2M : Extract the modulus of a complex data structure.
CMPLX2R : Extract the real part of a complex data structure.
CMPLXADD : Add two complex structures.
CMPLXCONJ : Produce the complex conjugate of a complex structure.
CMPLXDIV : Divide two complex structures.
CMPLXFILT : Create a mid-pass filter for complex data.

CMPLXMULT : Multiply two complex structures.
CMPLXSUB : Subtract two complex structures.
COSBELL : Create data that goes to zero at the edges in a cosine bell.
FFT : Take the forward FFT of a complex data structure.
I2CMPLX : Copy an array into the imaginary part of a complex structure.
PEAK : Determine the position of the highest peak in a spectrum.
R2CMPLX : Create a complex data structure from a real data array.
ROTX : Rotate data along the X-axis.

Fudging: **CSET** : Set regions of a spectrum to a constant value interactively.
DELOBJ : Delete a data object or a file.
ISEDIT : Allow interactive editing of a 1-d or 2-d spectrum.
LET : Assign a value to a named data object or variable.
LXSET : Set X array of a spectrum or image to a specified range.
LYSET : Set Y array of a spectrum or image to a specified range.
NCSET : Set a region of a spectrum to a constant.
RENOBJ : Change the name of a data object.
SPIED : Edit a spiketrum interactively.
TIPPEX : Modify individual pixel values with cursor.
XCADD : Add a constant to the X data in a file.
XCDIV : Divide the X data in a file by a constant.
XCMULT : Multiply the X data in a file by a constant.
XCSUB : Subtract a constant from the X data in a file.

Slices: **EXTLIST** : Add a number of non-contiguous lines in an image → a spectrum.
EXTRACT : Add contiguous lines of an image → a spectrum.
GROWX : Perform reverse function to that of EXTRACT.
GROWXT : Copy an image into contiguous XT planes of a cube.
GROWXY : Copy an image into contiguous XY planes of a cube.
GROWY : Perform reverse function to that of YSTRACT.
GROWYT : Copy an image into contiguous YT planes of a cube.
OPTEXTRACT : Extract a long slit spectrum using Horne's optimal extraction.
PROFILE : Determine a long slit spectrum profile for use by OPTEXTRACT.
SLICE : Take a slice with arbitrary end points through an image.
XTPLANE : Add contiguous XT planes of a data cube → an image.
XYPLANE : Add contiguous XY planes of a data cube → an image.
YSTRACT : Add contiguous columns of an image → a spectrum.
YTPLANE : Add contiguous YT planes of a data cube → an image.

Fibre images: **FINDSP** : Locate fibre spectra in an image.
POLEXT : Extract fibre spectra from an image after a FINDSP analysis.
OVERPF : Overlay a FINDSP fit on another image.

SPECTROMETERS — Fabry-Perot infra-red grating spectrometer: **FET321** : Extract a spectrum from 1 detector from etalon mode FIGS data.
FIGS321 : Process a FIGS data cube down to a single spectrum.
FIGS322 : Process a FIGS data cube down to an image.
FIGS422 : Process a FIGS image-mode hypercube down to an image.
FIGS423 : Process a FIGS image-mode hypercube down to a cube.
FIGS424 : Sort a FIGS image-mode hypercube into wavelength order.
FIGSEE : Generate a seeing ripple spectrum from a FIGS spectrum.
FIGSFLUX : Flux calibrate a FIGS spectrum.
IRCONV : Convert data in Janskys to W/m**2/um.
IRFLAT : Generate a ripple spectrum from an IR spectrum.

IRFLUX : Flux calibrate an IR spectrum using a black-body model.
REMBAD : Remove pixels that have been flagged as bad from data.

Echelle spectrometer: CDIST : Correct S-distortion using SDIST results.
ECHARC : Fit an echelle arc.
ECHFIND : Locate spectra in echelle data.
ECHMASK : Produce an extraction mask from an SDIST analysis.
ECHMERGE : Merge echelle spectra into a single long spectrum.
ECHSELECT : Select sky and object spectra for an echelle interactively.
ICUR : Show x,y and data values using ARGS cursor.
IMAGE : Display an image on the selected image display.
MASKEXT : Extract echelle orders using a mask created by ECHMASK.
ODIST : Overlay an SDIST fit on another image.
OFFDIST : Apply an offset to an SDIST fit.
SDIST : Analyse an image containing spectra for S-distortion.

ANALYSIS — Absorption line analysis: ABLINE : Analyse absorption lines interactively.
GAUSS : Fit Gaussians to emission or absorption lines interactively.

Photometry: CENTERS : Generate a file of object centroids from CPOS output.
CPOS : Select points with the image display cursor.
FOTO : Perform aperture photometry given CENTERS output.

Gaussian fitting to spectral lines: EMLT : Fit Gaussians to the strongest lines in a spectrum.
GAUSS : Fit Gaussians to emission or absorption lines interactively.

MISCELLANEOUS — Miscellaneous: CCDLIN : Apply a linearity correction to AAO CCD data.
ERRCON : Convert percentage error values to absolute values.
FIGSET : Set Figaro environment parameters, e.g. default directory.
RECOFF : Turn off logging of the terminal dialogue.
RECON : Turn on logging of the terminal dialogue.
RETYPE : Change the type of the main data array in a file.
SQRTERR : Generate an error array as Error = Square Root of (Data/Const).
TRIMFILE : Cut down the size of a Figaro file by removing any deadwood.
VSHOW : Display the values of Figaro user variables.

20.6 IRCAM — Infrared camera data reduction [SUN/41]

These are the IRCAM commands which are relevant specifically to data reduction:

Image Display — **AGAIN:** Plot the last image again.

CFLASH: Plot an image without scaling.

CHPLT: Plot PhaseA, PhaseB and Difference on workstation.

CNSIGMA: Plot an image with range N-sigma on mean at cursor posn.

CPLOT: Plot an image with user defined max,min with cursor.

CRANPLOT: Plot an image between user defined range using cursor.

CVARG: Plot an image with vargrey between max,min with cursor.

DISP: Plot STARE, SKY difference on workstation.

DISPHOT: Retired, use DISP.

FLASH: Plot an image without scaling.

MOREN: Plot an image using N_sigma range on mean again.

MOREP: Plot an image using plot between limits again.

MORER: Plot an image using ranplot range on mean again.

MOREV: Plot an image using vargrey CUT.

NSIGMA: Plot an image using N_sigma range on mean.

PLOT: Plot an image between maximum, minimum values.

PLOTGLITCH: Plot an image, glitchmarks it, plot result.

PLOTLOT: Plot and do stuff on a number of images.

RANPLOT: Plot an image between user defined range.

SHOW: Display an image with one of the plot procedures.

STPLT: Plot STARE, SKY difference on workstation.

VARGREY: Plot an image with variable scaling between max,min.

Colours — **ABLOCK:** Plot a colour block.

ALLCOL: Write all colour tables to image display.

CABLOCK: Plot a colour block at cursor position.

COL11: Write COL11 colour table.

COL13: Write COL13 colour table.

COL19: Write COL19 colour table.

COL3: Write COL3 colour table.

COL5: Write COL5 colour table.

COL7: Write COL7 colour table.

COLCYCLE: Cycle colour table n times into new colour table.

COLINV: Invert the colour table on subsequent COLTABs.

COLIST: List colour tables currently available.

COLOUR: Manipulate colours for PLT2D D-task.

COLTAB: Write a colour table.

COLTAB_LIST: List coltab tables with description.

CRECOLT: Create colour tables by pen/intensity selection.

GREY: Write a GREY colour table.

HEAT: Write the HEAT colour table.

MANYCOL: Combine n different colour tables into one output colour table.

PENCOL: Set the colour of a pen.

PENINT: Set the intensity of the 3 guns for a pen.

SPEC: Write a SPEC colour table.

WRITELUT: Write a colour table LUT to workstation.

Line Graphics — BORDER: Plot a border around last image.

BOX: Plot a box.

CBOX: Plot a box centred on cursor.

CC: Wrap around for CROSSCUT.

CCIRCLE: Plot a circle at cursor.

CCROSS: Plot a cross at cursor.

CCUT: Plot a cut using cursor defined points on current image.

CELLIPSE: Plot a ELLIPSE at cursor.

CIRCLE: Plot a circle.

CLINE: Plot a line between two cursor positions.

CONTOUR: Plot a contour image.

CONT_TIT: Set title for a contour plot.

CROSS: Plot a cross.

CROSSCUT: Plot an x and a y cut on current image.

CUT: Plot a cut on current image.

CUT2FF: Enable/disable writing of cut to ff file.

CUT_TIT: Set title for a cut plot.

ELLIPSE: Plot an ellipse.

FEATURE: Plot a line graphics/comment feature on workstation.

GRID: Plot a grid on an image.

HIST: Plot a histogram.

LINE: Plot a line between user specified positions.

LINECOL: Set all line graphics pens to specified color.

SURROUND: Put a border and ticks/numbers around image.

VEC: Plot a polarization vector plot on current workstation.

VEC_TIT: Set a title for a vector plot.

Text — LABEL: Label an image with position values from a file.

WRCCOM: Write a comment at cursor.

WRCOM: Write a comment on current workstation.

Maths — ADD: Add two images together.

CADD: Add a constant to an image.

CDIV: Divide an image by a constant.

CMULT: Multiply an image by a constant.

CSUB: Subtract a constant from an image.

DIV: Divide two images.

EXP10: Take 10 exponential of an image.

EXPE: Take e exponential of an image.

EXPON: Take N exponential of an image.

LOG10: Log base 10 of an image.

LOGAR: Log base N of an image.

LOGE: Log base e of an image.

MEDLOT: Subtract median from a number of images.

MULT: Multiply two images.

OADD: Procedure to run ADD rapi2d action.

OCADD: Procedure to run CADD rapi2d action.

OCDIV: Procedure to run CDIV rapi2d action.

OCMULT: Procedure to run CMULT rapi2d action.

OCSUB: Procedure to run CSUB rapi2d action.

ODIV: Procedure to run DIV rapi2d action.

OMULT: Procedure to run MULT rapi2d action.

OSUB: Procedure to run SUB rapi2d action.

POW: Raise image to arbitrary power.

ROOT: Take square root of an image.

SUB: Subtract two images.

TRIG: Perform trigonometrical functions on image.

Statistics — ANNSTATS: Annular aperture statistics with eccentric annuli.

APERADD: Perform simple aperture statistics.

APERPHOT: Aperture photometry program.

APPH1: Aperture photometry using cremap offset file.

HISTGEN: Generate histogram of an image (plot with HIST).

HISTO: Histogram statistics on an image.

MAG: Calculate magnitude of feature in image using input zp.

MSTATS: Statistics on n images through each pixel.

OCMAG: Procedure to run MAG rapi2d action with cursor.

OCSTATS: Procedure to run STATS rapi2d action with cursor.

OHISTGEN: Procedure to run HISTGEN A-task

OHISTO: Procedure to run HISTO rapi2d action.

OSTATS: Procedure to run STATS rapi2d action.

STATS: Statistics on sub-image.

Image size changing — ABSEP: Separate odd and even readout channels into 2 images.

BINUP: Reduce image in size by binning pixels (separate x and y).

COMPADD: Compress an image in size by adding pixels.

COMPAVE: Compress an image in size by averaging pixels.

COMPICK: Compress an image in size by picking pixels.

COMPRESS: General compression in size program.

DISPICK: Procedure to run PICKIM rapi2d action with cursor.

MANIC: Change an image size arbitrarily.

OEFIX: Scale odd and even channels wrt median in channels.

OSHSIZE: Procedure to run SHSIZE rapi2d action.

PICKIM: Pixeks a sub-image from an image and stores.

PICKLOT: Pickims a number of images.

PIXDUPE: Increase image size by pixel duplication.

SHSIZE: Return size of an image.

SQORST: Change shape and dimensionality of an image.

XGROW: Grow an image in x direction.

YADD: Add up all pixels down each column and generate linear output.

YGROW: Grow an image in y direction.

Mosaicing — AUTOMOS: Correct dc offsets and mosaic n images automatically.

CREQUILT: Create a quilt file from lists of offsets and images.

MOFF: Calculate spatial and dc offset of 2 overlapping images.

MOS2: Display 2 images, gets stats and scales and mosaics.

MOSAIC: Mosaic together n overlapping images.

MOSAIC2: Mosaic 2 overlapping images together.

MOSCOR: Calculate dc offset between two offset images.

QUILT: Mosaic n images together using terminal/file input.

WMOSAIC: Mosaic with weighting on image contributions.

WQUILT: Quilt with weighting on image contributions.

Polarimetry — APERPOL: Aperture polarimetry from 4 input intensity images.

DEVFCS: Calculate deviation of polarization from centro-symmetry.

FLATPOL: Flat-field and deglitch 4 polarization images.

OCAPERPOL: Procedure to run APERPOL action with cursor.

POLCAL: Calculate polarization from input 4 intensity images.

POLLY: Calculate polarization from input 4 values.

POLREG: Move 4 polarization images to average position and trim.

POLSHFT: Move 4 polarization images to average position.

POLSHIFT: Wrap around for POLSHFT.

POLSHOT: Remove shot-noise polarization contribution from image.

POLTHRESH: Threshold 4 polarization intensity images.

SKYSUB4: Subtract median of area of 4 images image from images.

THETAFIX: Correct polarization position angle to 0-180 degs.

Daophot — DAOCEN: Take xcursor input and calculate centroid at that position.

DAOFIND: Plot crosses or circles on each star found.

DAOGID: Get ID of nearest star to cursor selected one.

DAOGID2: Get ID of nearest star to cursor selected one.

Inquiries — GETCP: Get the pixel position of a cursor input.

GETCR: Get the real position of a cursor input.

GETMC: Get the calculated maximum,minimum in a PLOT/NSIGMA.

GETMM: Get the maximum and minimum specified for a PLOT.

GETPEN: Get pens of all line graphics and pen numbers.

GETPLOT: Get the current state of image area PLOT.

GETPOLCOL: Get the current settings of the colours for polax.

GETPS: Get the plate scale for feature plotting on images.

GETRAST: Get the size in raster units of the current workstation.

GETSTEN: Get the start and size in raster units of image area.

Set — SETAREA: Copy useful files from standard directory to users.

SETBAD: Set bad pixel image to be used.

SETCEN: Set the centre of the image in a PLOT/NSIGMA.

SETCIM: Set the cursoring image to other than that displayed.

SETCOL: Set the colour of special features.

SETCOMORI: Change the character orientation.

SETCONT: Set up the contour plot.

SETCONTIC: Set contour maps tick mark extent.

SETCUR: Set where the cursor refers to in CPLOT, CNSIGMA etc.

SETCURMARK: Set the cursor marking parameter from user's choice.

SETCUT: Set up a cut plot.

SETCUTAXRAT: Set the axis ratio for cut.

SETDARK: Set dark image to be used.

SETFONT: Change the font and character orientation.

SETMAG: Set the magnification for subsequent PLOTS/NSIGMAS.

SETMAX: Set the maximum for image scaling in a PLOT.

SETMIN: Set the minimum for image scaling in a PLOT.

SETMM: Set the maximum and minimum for image scaling in a PLOT.

SETNUM: Set offset or ra/dec mode for contour maps and surround.

SETNUMORI: Set the orientation of numbers in around image.

SETNUMSCA: Set the numbers scale factor for around image.

SETPLOTAREA: Set the option to plot whole or sub- image.

SETPOLCOL: Set the current settings of the colours for polax.

SETPRE: Set correct file prefix for observation files.

SETPS: Set the plate scale for feature plotting on image.

SETQUAD: Set a quadrant for an image plot.

SETRADEC: Set ra/dec for contour maps and surround.

SETUP_IRCAM: Set IRCAM parameters for IRACS,MOTASK,PLT2D D_tasks.

SETVAL: Set a specified value in an image at user locations.

SETVARG: Set X and Y percentage cut for vargrey plot.

SETVEC: Set up a polarization vector map on current workstation.

Bad Pixel Removal — CHPIX: Change pixel values in image.

DEGLOT: Deglitch a number of images.

GLITCH: Remove bad pixels by interpolation.

GLITCHMARK: Mark glitches for glitch A-task interactively.

INSETB: Set pixels inside user defined box to user value.

INSETC: Set pixels inside user defined circle to user value.

OGLITCH: Procedure to run GLITCH rapi2d action.

OUTSETB: Set pixels outside specified box to user value.

OUTSETC: Set pixels outside user specified circle to user value.

Smoothing — BLOCK: Block smooth an image.

GAUSS: Gaussian smooth an image.

GAUSSTH: Gaussian smooth images below a specified threshold.

LAPLACE: Laplace transform an image.

MEDIAN: Spatially median filter an image.

General and Miscellaneous — AMCORR: Correct images for air mass.

ASCIIFILE: Take text/variables from ADAMCL and write to ASCII file.

ASCIILIST: Convert SDF image to ASCII list of numbers.

BATCH_ZEROPOINTS: Read zeropoint for DISP from ASCII file.

CALMAG: Calculate magnitude from number and zeropoint.

CALZER: Calculate zeropoint from number and magnitude.

CENTROID: Calculate centroid of stellar profile.

CIT: Wrap around for CLEARIT.

CLEAR: Clear the workstation.

CLEARIT: Clear a section of the image display screen.

CLOSE: Close plotting.

COLMED: Calculate median down columns and create output mask.

CREFP: Create fp scan list in SDF format.

CREFRAME: Create an image with standard patterns.

CREMAP: Create image positional offset list in SDF format (gomos).

CURSOR: Display the cursor and returns X,Y, value.

DARKLOT: Dark subtract a number of images.

DARKSUB: Subtract dark from a number of images.

DATASW: Switch data input from numbers to names.

DEFGRAD: Define gradients in image and return map.

DEFPROMPT: Define the prompt for ADAMCL.

DIST: Calculate distance and angle between two points in image.

FCOADD: Add up n images into one image.

FITSREAD: Read fits tape and create SDF file.

FITSWM: Write SDF files to fits tape.

FLATDEG: Flat field and deglitch a number of images.

FLATLOT: Flat field a number of images.

FLATRED: Reduce image.

FLIP: Flip an image east-west or north-south.

GFIT: Fit Gaussian to star.

HARDCOPY: Close plotting, open it on hardcopy workstation.

HELPLIST: List commonly used procedures.

HISTEQ: Histogram equalization of an image.

INDEX: Index through an SDF image listing components.

INTLK: Generate integer map of an image.

LINCONT: Linearize normal readout (non-NDR) images.

LINCONT_NDR: Linearize NDR readout images.

LISTMAP: List out a mosaic offset file to terminal.

LOOK: Inspect pixels in an image.

LOUD: Set the bell parameters to normal values.

MCURSOR: Display repeatedly the cursor.

MED3D: Median filter through stack of n images.

NUMB: Calculate number of pixels above specified signal.

OBSEXT: Extract an image from a observation container file.

OBSLIST: List a specified component of an observation structure.

ODIST: Procedure to run DIST rapi2d action.

OLOOK: Procedure to run LOOK rapi2d action.

ONUMB: Procedure to run NUMB rapi2d action.

OPEN: Open plotting on a workstation.

PLT2D_RESET: Close plotting and open it again.

RADIM: Generate radial image profile from image and centre position.

REDLIST: List reduction routines for info.

RELOAD_PLT2D: Load/reload Plt2d.

RELOAD_RAPI2D: Load/reload Rapi2d.

ROTATE: Rotate an image by an arbitrary amount.

ROWMED: Median filter along rows of an image generating a map.

SHADOW: Shadow enhance an image.

SHIFT: Shift an image by fraction pixels by linear interpolation.

SHIFT2: Move 2 images to average position.

SHIFT3: Centroid and shift 3 images.

SNAPFILE: Convert spanshot images to SDF format images.

SNOOP_IRCAM: Get IRCAM parameters from IRACS,MOTASK,PLT2D.

SOFT: Set the bell parameters to 0 values.

SQ: Wrap around for SETQUAD.

STARFIT: Fit star and return stack dump (Starlink routine).

STD: Use cursor to get star position and aperadd.

STDPLOT: Plot last GKS QMS file to laser printer.

STEPIM: Step an image (like pseudo-contouring with values filled).

SUBDARK: Subtract dark.

SYS: Get the SYS/S from VMS.

THRESH: Set values above and below specified levels to user values.

THRESH0: Set values below 0 to user value.

TOMAG: Convert intensity to magnitudes in images.

TRACE: Trace HDS structure.

TRANDAT: Transfer ASCII list of numbers to SDF image.

USE: Get the users from VMS.

20.7 KAPPA — Kernel applications [SUN/95]

I/O — Image generation and input:

CREFRAME : Generate a test 2-d data array from a selection of several types.

FITSDIN : Read a FITS disk file composed of simple, group or table files.

FITSIMP : Import FITS information into an NDF extension.

FITSIN : Read a FITS tape composed of simple, group or table files.

TRANDAT : Convert free-format data into an NDF.

DISPLAY — Detail enhancement:

HISTEQ : Perform an histogram equalisation on a 2-d data array.

LAPLACE : Perform a Laplacian convolution as an edge detector in a 2-d data array.

SHADOW : Enhance edges in a 2-d data array using a shadow effect.

THRESH : Create a thresholded version of a data array (new values for pixels outside defined thresholds are specified).

THRESH0 : Create a thresholded version of a data array (every value outside the defined thresholds is set to zero).

Display control:

BLINK : Blink two planes of an image display.

CURSOR : Report co-ordinates of points selected using cursor and select current picture.

GDCLEAR : Clear a graphics device and purge its database entries.

GDSTATE : Show the current status of a graphics device.

IDCLEAR : Clear an image display and purge its database entries.

IDINVISIBLE : Make memory planes of an image-display device invisible.

IDPAN : Pan and zoom an `ARGS` or an `IKON`.

IDPAZO : Pan and zoom an image-display device.

IDRESET : Perform a hardware reset of an `ARGS` or an `IKON`.

IDSTATE : Show the current status of an image display.

IDUNZOOM : Unzoom and re-centre an image-display device.

IDVISIBLE : Make all the memory planes of an image-display device visible.

PICDEF : Define a new graphics-database picture or an array of pictures.

PICIN : Find the attributes of a picture interior to the current picture.

PICLABEL : Label the current graphics-database picture.

PICLIST : List the pictures in the graphics database for a device.

PICSEL : Select a graphics-database picture by its label.

Device selection:

GDNAMES : Show which graphics devices are available.

GDSET : Select a current graphics device.

IDSET : Select a current image-display device.

OVSET : Select a current image-display overlay.

Lookup/Colour tables:

CRELUT : Create or manipulate an image-display lookup table using a palette.

LUTABLE : Manipulate an image-display colour table.

LUTBGYRW : Load the *BGYRW* lookup table.

LUTCOL : Load the standard colour lookup table.

LUTCONT : Load a lookup table to give the display the appearance of a contour plot.

LUTFC : Load the standard false-colour lookup table.

LUTFLIP : Flip the colour table of an image-display device.

LUTGREY : Load the standard greyscale lookup table.

LUTHEAT : Load the *heat* lookup table.
LUTHILITE : Highlight a colour table of an image-display device.
LUTIKON : Load the *Ikon* lookup table.
LUTNEG : Load a negative greyscale lookup table.
LUTRAMPS : Load the coloured-ramps lookup table.
LUTREAD : Load an image-display lookup table from an NDF.
LUTROT : Rotate the colour table of an image-display device.
LUTSAVE : Save the current colour table of an image-display device in an NDF.
LUTSPEC : Load a spectrum-like lookup table.
LUTTWEAK : Tweak a colour table of an image display.
LUTVIEW : Draw a colour-table key.
LUTZEBRA : Load a pseudo-contour lookup table.
TWEAK : Adjust a colour table interactively.

Output:

COLUMNAR : Draw a perspective-histogram representation of a 2-d NDF.
CONTOUR : Contour a 2-d NDF.
CONTOVER : Contour a 2-d data array overlaid on an image displayed previously.
DISPLAY : Display a 2-d NDF.
GREYPLOT : Produce a greyscale plot of a 2-d NDF.
HIDE : Draw a perspective plot of a 2-d NDF.
INSPECT : Inspect a 2-d NDF in a variety of ways.
LINPLOT : Draw a line plot of a 1-d NDF's data values against their axis co-ordinates.
LOOK : Output the values of a sub-array of a 2-d data array to the screen or an ASCII file.
MLINPLOT : Draw a multi-line plot of a 2-d NDF's data values v. axis co-ordinates.
SNAPSHOT : Dump an image-display memory to hardcopy and, optionally, to an NDF.
TURBOCONT : Contour a 2-d NDF quickly.

Palette:

PALDEF : Load the default palette to a colour table.
PALENTRY : Enter a colour into an image display's palette.
PALREAD : Fill the palette of a colour table from an NDF.
PALSAVE : Save the current palette of a colour table to an NDF.

MANIPULATION — Arithmetic:

ADD : Add two NDF data structures.
CADD : Add a scalar to an NDF data structure.
SUB : Subtract one NDF data structure from another.
CSUB : Subtract a scalar from an NDF data structure.
MULT : Multiply two NDF data structures.
CMULT : Multiply an NDF by a scalar.
DIV : Divide one NDF data structure by another.
CDIV : Divide an NDF by a scalar.
EXP10 : Take the base-10 exponential of each pixel of a data array.
EXPE : Take the exponential of each pixel of a data array (base *e*).
EXPON : Take the exponential of each pixel of a data array (specified base).
POW : Take the specified power of each pixel of a data array.
LOG10 : Take the base-10 logarithm of each pixel of a data array.
LOGAR : Take the logarithm of each pixel of a data array (specified base).
LOGE : Take the natural logarithm of each pixel of a data array.
MATHS : Evaluate mathematical expressions applied to NDF data structures.
TRIG : Perform a trigonometric transformation on a data array.

Pixel editing:

CHPIX : Replace the values of selected pixels in a 2-d data array with user-specified values.
GLITCH : Replace bad pixels in a 2-d data array with the local median.
NOMAGIC : Replace all magic-value pixels in a data array by a user-defined value.
OUTSET : Set pixels outside a specified circle in a 2-d data array to a specified value.
SEGMENT : Copy polygonal segments from one 2-d data array to another.
SETMAGIC : Replace all pixels with given value in data array by magic value.
ZAPLIN : Replace regions in a 2-d data array by bad values or by linear interpolation.

Configuration change:

FLIP : Reverse an NDF's pixels along a specified dimension.
INSPECT : Inspect a 2-d NDF in a variety of ways.
MANIC : Convert all or part of a data array from one dimensionality to another.
PICK2D : Create a new 2-d data array from a subset of another.
ROTATE : Rotate a 2-d data array through any angle.
SHIFT : Realign a 2-d data array via an x,y shift.

Combination:

MOSAIC : Merge several non-congruent 2-d data arrays into one output data array.
NORMALIZE : Normalize NDF to similar NDF by calculating scale and zero difference.
QUILT : Generate a mosaic from equally sized 2-d data arrays.

Compression and Expansion:

COMPADD : Reduce the size of a 2-d data array by adding neighbouring pixels.
COMPAVE : Reduce the size of a 2-d data array by averaging neighbouring pixels.
COMPICK : Reduce the size of a 2-d data array by picking every n^{th} pixel.
COMPRESS : Reduce the size of a 2-d data array by averaging neighbouring pixels by different amounts in x and y.
PIXDUPE : Expand a 2-d data array by pixel duplication.
SQORST : Squash or stretch a 2-d data array in either or both axes.

Filtering:

BLOCK : Smooth a 2-d image using a square or rectangular box filter.
FOURIER : Perform forward and reverse Fourier transforms on 2-d NDFs.
GAUSS : Smooth a 2-d image using a symmetrical Gaussian filter.
MEDIAN : Smooth a 2-d data array using a 2-d weighted median filter.
MEM2D : Perform a Maximum-Entropy deconvolution of a 2-d NDF.
SURFIT : Fit a polynomial or spline surface to a 2-d data array.

NDF and HDS components:

ERASE : Erase an HDS object.
FITSIMP : Import FITS information into an NDF extension.
SETBB : Set a new value for the quality bad-bits mask of an NDF.
SETLABEL : Set a new label for an NDF data structure.
SETTITLE : Set a new title for an NDF data structure.
SETTYPE : Set a new numeric type for the data and variance components of an NDF.
SETUNITS : Set a new units value for an NDF data structure.
SETVAR : Set new values for the variance component of an NDF data structure.

ANALYSIS — Statistics:

APERADD : Derive statistics of pixels within a specified circle of a 2-d data array.
HISTAT : Generate an histogram of a 2-d data array to compute statistics.
HISTOGRAM : Derive histograms of sub-arrays within a 2-d data array.
INSPECT : Inspect a 2-d NDF in a variety of ways.
MSTATS : Cumulative statistics on a 2-d sub-array over a sequence of 2-d data arrays.
NUMB : Count the elements of an array with values greater than a specified value.

NUMBA : Count the elements of an array with absolute values greater than specified.
STATS : Compute simple statistics for an NDF's pixels.
STATS2D : Compute simple statistics for a 2-d data array.

Other:

CENTROID : Find the centroids of star-like features in an NDF.
NORMALIZE : Normalize NDF to similar NDF by calculating scale and zero difference.
PSF : Determine the parameters of a model star profile by fitting star images in a 2-d NDF.
SURFIT : Fit a polynomial or spline surface to 2-d data array.

INQUIRIES and STATUS — **GLOBALS** : Display the values of the KAPPA global parameters.
FITSLIST : List the FITS extension of an NDF.
NDFTRACE : Display the attributes of an NDF data structure.

MISCELLANEOUS — **KAPHELP** : Give help about KAPPA.

20.8 PHOTOM — Aperture photometry [SUN/45]

A — Annulus: Alter the way in which the background level is measured.

C — Centroid: Specify if the object is centered in the aperture before doing the measurement.

E — Exit: Terminate the current PHOTOM session.

F — File of positions: Do measurements automatically.

H — Help: Display a line of help information for each command.

I — Interactive shape: Adjust the size and shape of the cursor interactively.

M — Measure: Measure individually selected objects interactively.

N — Non-interactive shape: Specify the size and shape of the aperture from the keyboard.

O — Options: Change values of parameters specified in the interface file from the keyboard.

P — Photon statistics: Alter the way in which errors are calculated.

S — Sky: Choose the method of estimating the background level in the sky aperture.

V — Values: Display the current settings of significant parameters.

20.9 PISA — Object finding and analysis [SUN/109]

ADDNOISE: Add Poissonian or Gaussian noise to model (or any other) data.

PISA2SCAR: Convert PISAFIND and PISAPEAK result files to SCAR format, so they can be used either by SCAR or by the CHA catalogue manipulation applications.

PISACUT: Separate a file of variables into two groups by thresholding the values in a single column.

PISAFIND: Perform image analysis on a 2-d data frame. There are two basic modes of operation — *isophotal analysis* (in which pixels with data values above the threshold are examined for connectivity and combined into objects) and *profile fitting* (in which an analytical stellar profile is fitted to the objects found by a preliminary isophotal analysis).

PISAFIT: Fit the radially symmetric mixed Gaussian/Exponential/Lorentzian function, as used by PISAFIND in its profile fitting mode, to objects whose **accurate** positions are given in a formatted list.

PISAGEN: Generate model data using the PISA profile fitting function using the positions and integrated intensities found by PISAFIND in its profile fitting mode (or any other data in similar format).

PISAKNN: Use the results of PISAPEAK to discriminate objects into two classes using KNN (k nearest neighbours) distribution-free multivariate discrimination to classify objects into two classes.

PISAMATCH: Match the indices in one file against those in a second file, writing the matching entries of the second file into an output file.

PISAPEAK: Transform the PISAFIND parameterisations so that the variables are intensity invariant, assuming a stellar profile, for use in star-galaxy separation.

PISAPLOT: Plot the results of the PISAFIND analysis as a series of ellipses which reflect the size and shape of the objects as defined by their parameters.

20.10 SCAR — Star catalogue database system [SUN/70,106]

Database management — SEARCH: Select subsets.

SORT: Reorder.

JOIN: Find objects in common.

DIFFER: Find objects not in common.

MERGE: Merge catalogues.

SPLIT: Split into two catalogues.

WITHIN: Select objects from within a polygon.

CONVERT: Change format, make indexes, new fields etc.

EDIT: Add and delete records, and correct values.

Display — REPORT: List all or some of a catalogue's contents.

PRINT: Print it to a default file.

LISTOUT: Show values of a given field.

CALC: Calculate new fields.

RECALC: Update existing fields.

CONVERT: Change format.

WRAP: Produce a printable version of a catalogue with records longer than 132 characters.

CHART: Plot objects on a finding chart.

AITOFF: Plot all objects on an all-sky plot.

IMAGEPLOT: Plot objects for COSMOS-style processing.

Statistics — LITTLEBIG: Find either the largest or smallest numbers in a catalogue.

SAMPLE: Select every Nth object from a catalogue.

HISTOGRAM: Plot a histogram of a given field.

SCATTER: Plot two fields and perform regression analysis.

CORRELATE: Correlate (non-parametrically) two fields.

LINCOR: Compute the Pearson product-moment linear correlation coefficient.

Create and process description files — EXTAPE: Examine the contents of an ASCII tape and dump it to disk.

LISTIN: Read a simple VMS file into SCAR.

FORM1: Create a description file in screen mode.

FORM2: Create a description file in prompt mode.

ASPIC: Create a description file for an ASPIC/IAM catalogue.

ASCII: Convert a description file for BINARY data to one for ASCII data.

BINARY: Convert a description file for ASCII data to one for BINARY data.

POLYGON: Create a description file of polygon vertices.

Small tools — SETUP: Define default values for certain options.

CATSIZE: Find the number of objects in a catalogue.

COUNTREC: Count the number of objects in a sequential file.

COSMAGCAL: Calculate magnitudes from COSMOS magnitudes.

HARDCOPY: Produce hardcopy of output produced by graphics programs.

MOUNT: Allocate a deck, mount a foreign tape, and assign it a logical name.

DISMOUNT: Dismount a tape, deallocate a deck, and deassign the logical name.

DSCFHELP: Insert a description file into a help library.

DEBUG: Switch on the VMS debugger (for Programmers only).

GETPAR: Get a parameter value when in the ICL environment.

20.11 SPECDRE — Spectroscopy data reduction [SUN/140]

I/O — ASCIN: Read a 1-d or N-d data set from an ASCII table.

ASCOUT: Write a subset to an ASCII table.

Display — SPECPLOT: Plot a spectrum.

Data manipulation — GOODVAR: Replace negative, zero and bad variance or error values.

Statistics — CORREL: Correlate three data sets.

Reshaping — GROW: Copy from an N-d cube into an (N+M)-d one.

SUBSET: Take a subset of a data set (up to 10-d).

XTRACT: Average an N-d cube into an (N-M)-d one.

Data calibration — BBODY: Calculate a black body spectrum.

Fitting — FITGAUSS: Fit continuum and Gaussian to a spectrum.

20.12 SST — Simple software tools [SUN/110]

FORSTATS: Analyse a sequence of Fortran 77 source files, divide their contents into program units, and produce statistics about the number and distribution of code and comment lines in each unit. Compare these statistics are with typical values from well-crafted Fortran code.

PROCVT: Convert 'old-style' ADAM/SSE routine prologues into the format generated by the extended VAX Language Sensitive Editor (STARLSE).

PROHLP: Read a series of Fortran 77 source files containing prologues generated by STARLSE, and produce an output file for each routine containing user-documentation in a format suitable for insertion into a Help library.

PROLAT: Read a series of Fortran 77 source files containing prologues generated by STARLSE, and produce an output file containing user-documentation for each routine written in LaTeX.

PROPAK: Read a series of Fortran 77 source files containing prologues generated by STARLSE, and produce an output file containing an LSE 'package definition' suitable for use with STARLSE.

20.13 TSP — Time-series and polarimetry analysis [SUN/66]

Data I/O — BUILD3D: Insert a Figaro frame into a time series image.

RCCDTS: Read AAO CCD time series data.

RCGS2: Read CGS2 polarimetry data.

RFIGARO: Read a Stokes parameter spectrum from a Figaro image.

RHATPOL: Read Hatfield polarimeter data.

RHATHSP: Read Hatfield polarimeter high speed photometry data.

RHDSPLOT: Read ASCII files of Hatfield polarimeter data.

RHSP3: Read an HSP3 tape.

RIRPS: Read IRPS photometry data.

RTURKU: Read ASCII files of data from the Turku UBVRI polarimeter.

TLIST: List time series data to a file.

XCOPY: Copy wavelength data from a Figaro spectrum.

Data analysis — CALFIT: Fit a calibration curve to a polarization spectrum.

CALFITPA: Fit a calibration curve to the polarization position angle.

CALIB: Efficiency calibrate a polarization spectrum.

CALPA: Position angle calibrate a polarization spectrum.

CCD2POL: Reduce CCD spectropolarimetry data.

CCD2STOKES: Reduce CCD spectropolarimetry data.

CCDPHOT: Photometry of a star on a time series image.

CCDPOL: Polarimetry of a star on a time series image.

CMULT: Multiply a polarization spectrum by a constant.

COMBINE: Combine two polarization spectra.

DIVIDE: Divide a polarization spectrum by an intensity spectrum.

DSTOKES: Delete a Stokes parameter from a dataset.

EXTIN: Correct a polarization spectrum for extinction.

FLCONV: Convert a flux calibrated spectrum to f-lambda.

FLIP: Invert the sign of the Stokes parameter in a spectrum.

IMOTION: Analyze the image motion in a time series image.

IPCS2STOKES: Reduce IPCS spectropolarimetry data.

LHATPOL: List Hatfield polarimeter infrared data.

LMERGE: Merge two polarization spectra.

LTCORR: Apply light time corrections to the time axis of a data set.

PTHETA: Output the P and Theta values for a polarization spectrum.

QUMERGE: Merge Q and U spectra into single dataset.

QUSUB: Subtract a Q,U vector from a polarization spectrum.

REVERSE: Reverse a spectrum in the wavelength axis.

ROTPA: Rotate the position of a polarization spectrum.

SCRUNCH: Rebin a polarization spectrum.

SHIFTADD: Add frames of time series image, correcting for image motion.

SPFLUX: Apply flux calibration to a polarization spectrum.

SUBSET: Take a subset of a dataset in wavelength or time axes.

SUBTRACT: Subtract two polarization spectra.

TBIN: Bin a time series.

TDERIV: Calculate time derivative of a dataset.

TEXTIN: Correct a time series dataset for extinction.

TMERGE: Merge two time series datasets.

TSETBAD: Mark bad points in time series interactively.

TSEXTRACT: Extract optimally a light curve from a time series image.

TSHIFT: Apply a time shift to a dataset.

TSPROFILE: Determine a spatial profile from a time series image.

Plotting — DISPLAY: Display a 3-d TSP dataset on an image display device.

EPLOT: Plot a polarization spectrum as P, Theta with error bars.

FPLOT: Plot a polarization spectrum as polarized intensity.

PHASEPLOT: Plot time series data against phase.

PPLOT: Plot a polarization spectrum as P, Theta.

QPLOT: Plot time series data quickly.

QUPLOT: Plot a polarization spectrum in the Q,U plane.

SPLOT: Plot a polarization spectrum with a single Stokes parameter.

TSPLOT: Plot time series data.

Chapter 21
Subroutine Libraries

This chapter lists the names of the subroutines in the libraries mentioned in Chapter 13, together with brief descriptions of their functions. Its purpose is to help you find quickly what functions are *available*. It does not specify argument lists — this would have doubled the size of the chapter and made it much more cluttered. Argument lists are specified in the original documentation, which is referenced on the title line.

21.1 Parameter system

21.1.1 PAR — Parameter system [APN/6]

The parameter system routines are listed below. The letter '**d**' stands for the dimensionality:

0 — scalar.

1 — vector.

N — n-d array.

V — vectorised.

The letter '**t**' stands for one of the five HDS data types:

_INTEGER

_REAL

_DOUBLE

_LOGICAL

_CHAR

There are no PAR_DEFVx routines.

PAR_CANCL: Cancel a parameter.

PAR_DEFdt: Set a dynamic default parameter value.

PAR_GETdt: Read a parameter value.

PAR_PROMT: Set a new prompt string for a parameter.

PAR_PUTdt: Write a parameter value.

PAR_STATE: Return the state of a parameter.

21.2 Data system

21.2.1 NDF — NDF data structure access [SUN/33]

Parameter — **NDF_ASSOC:** Associate an existing NDF with a parameter.

NDF_CINP: Obtain an NDF character component value via the parameter system.

NDF_CREAT: Create a new simple NDF via the parameter system.

NDF_CREP: Create a new primitive NDF via the parameter system.

NDF_EXIST: See if an existing NDF is associated with a parameter.

NDF_PROP: Propagate NDF information to create a new NDF via the parameter system.

Message — **NDF_CMSG:** Assign the value of an NDF character component to a message token.

NDF_MSG: Assign the name of an NDF to a message token.

Accessing existing NDFs — **NDF_ASSOC:** Associate an existing NDF with a parameter.

NDF_EXIST: See if an existing NDF is associated with a parameter.

NDF_FIND: Find an NDF in an HDS structure and import it into the NDF system.

NDF_IMPRT: Import an NDF into the NDF system from HDS.

Inquiring NDF attributes — **NDF_BOUND:** Inquire the pixel-index bounds of an NDF.

NDF_DIM: Inquire the dimension sizes of an NDF.

NDF_ISACC: Determine whether a specified type of NDF access is available.

NDF_ISBAS: Inquire if an NDF is a base NDF.

NDF_ISTMP: Determine if an NDF is temporary.

NDF_NBLOC: Determine the number of blocks of adjacent pixels in an NDF.

NDF_NCHNK: Determine the number of chunks of contiguous pixels in an NDF.

NDF_SAME: Inquire if two NDFs are part of the same base NDF.

NDF_SIZE: Determine the size of an NDF.

NDF_VALID: Determine whether an NDF identifier is valid.

Inquiring component attributes — **NDF_BAD:** Determine if an NDF array component may contain bad pixels.

NDF_BB: Obtain the bad-bits mask value for the quality component of an NDF.

NDF_CLEN: Determine the length of an NDF character component.

NDF_CMPLX: Determine whether an NDF array component holds complex values.

NDF_FORM: Obtain the storage form of an NDF array component.

NDF_FTYPE: Obtain the full data type of an NDF array component.

NDF_QMF: Obtain the value of an NDF's quality masking flag.

NDF_STATE: Determine the state of an NDF component (defined or undefined).

NDF_TYPE: Obtain the numeric data type of an NDF array component.

Creating and deleting NDFs — **NDF_CREAT:** Create a new simple NDF via the parameter system.

NDF_CREP: Create a new primitive NDF via the parameter system.

NDF_DELET: Delete an NDF.

NDF_NEW: Create a new simple NDF.

NDF_NEWP: Create a new primitive NDF.

NDF_PROP: Propagate NDF information to create a new NDF via the parameter system.

Setting NDF attributes — **NDF_NOACC:** Disable a specified type of access to an NDF.

NDF_SBND: Set new pixel-index bounds for an NDF.

NDF_SHIFT: Apply pixel-index shifts to an NDF.

Setting component attributes — NDF_RESET: Reset an NDF component to an undefined state.

NDF_SBAD: Set the bad-pixel flag for an NDF array component.

NDF_SBB: Set a bad-bits mask value for the quality component of an NDF.

NDF_SQMF: Set a new logical value for an NDF's quality masking flag.

NDF_STYPE: Set a new type for an NDF array component.

Accessing component values — NDF_CGET: Obtain the value of an NDF character component.

NDF_CPUT: Assign a value to an NDF character component.

NDF_MAP: Obtain mapped access to an array component of an NDF.

NDF_MAPQL: Map the quality component of an NDF as an array of logical values.

NDF_MAPZ: Obtain complex mapped access to an array component of an NDF.

NDF_QMASK: Combine an NDF quality value with a bad-bits mask to give a logical result.

NDF_UNMAP: Unmap an NDF or a mapped NDF array.

Inquiring and setting axis attributes — NDF_ACLEN: Determine the length of an NDF axis character component.

NDF_ACRE: Ensure that an axis coordinate system exists for an NDF.

NDF_AFORM: Obtain the storage form of an NDF axis array.

NDF_ANORM: Obtain the logical value of an NDF axis normalisation flag.

NDF_AREST: Reset an NDF axis component to an undefined state.

NDF_ASNRM: Set a new value for an NDF axis normalisation flag.

NDF_ASTAT: Determine the state of an NDF axis component (defined or undefined).

NDF_ASTYP: Set a new numeric type for an NDF axis array.

NDF_ATYPE: Obtain the numeric type of an NDF axis array.

Accessing axis values — NDF_ACGET: Obtain the value of an NDF axis character component.

NDF_ACMSG: Assign the value of an NDF axis character component to a message token.

NDF_ACPUT: Assign a value to an NDF axis character component.

NDF_AMAP: Obtain mapped access to an NDF axis array.

NDF_AUNMP: Unmap an NDF axis array component.

Creating and controlling identifiers — NDF_ANNUL: Annul an NDF identifier.

NDF_BASE: Obtain an identifier for a base NDF.

NDF_BEGIN: Begin a new NDF context.

NDF_CLONE: Clone an NDF identifier.

NDF_END: End the current NDF context.

NDF_VALID: Determine whether an NDF identifier is valid.

Handling NDF (and Array) sections — NDF_BASE: Obtain an identifier for a base NDF.

NDF_BLOCK: Obtain an NDF section containing a block of adjacent pixels.

NDF_CHUNK: Obtain an NDF section containing a chunk of contiguous pixels.

NDF_NBLOC: Determine the number of blocks of adjacent pixels in an NDF.

NDF_NCHNK: Determine the number of chunks of contiguous pixels in an NDF.

NDF_SECT: Create an NDF section.

NDF_SSARY: Create an array section, using an NDF section as a template.

NDF_XIARY: Obtain access to an array stored in an NDF extension.

Matching and merging attributes — NDF_MBAD: Merge the bad-pixel flags of the array components of a pair of NDFs.

NDF_MBADN: Merge the bad-pixel flags of the array components of a number of NDFs.

NDF_MBND: Match the pixel-index bounds of a pair of NDFs.

NDF_MBNDN: Match the pixel-index bounds of a number of NDFs.

NDF_MTYPE: Match the types of the array components of a pair of NDFs.

NDF_MTYPN: Match the types of the array components of a number of NDFs.

Creating placeholders — NDF_PLACE: Obtain an NDF placeholder.

NDF_TEMP: Obtain a placeholder for a temporary NDF.

Copying NDFs — NDF_COPY: Copy an NDF to a new location.

NDF_PROP: Propagate NDF information to create a new NDF via the parameter system.

Handling extensions — NDF_XDEL: Delete a specified NDF extension.

NDF_XGT0t: Read a scalar value from a component within a named NDF extension.

NDF_XIARY: Obtain access to an array stored in an NDF extension.

NDF_XLOC: Obtain access to a named NDF extension via an HDS locator.

NDF_XNAME: Obtain the name of the Nth extension in an NDF.

NDF_XNEW: Create a new extension in an NDF.

NDF_XNUMB: Determine the number of extensions in an NDF.

NDF_XPT0t: Write a scalar value to a component within a named NDF extension.

NDF_XSTAT: Determine if a named NDF extension exists.

Tuning — NDF_GTUNE: Obtain the value of an NDF system tuning parameter.

NDF_TRACE: Set the internal NDF system error-tracing flag.

NDF_TUNE: Set an NDF system tuning parameter.

21.2.2 ARY — ARRAY data structure access [SUN/11]

Message — ARY_MSG: Assign the name of an array to a message token.

Accessing existing arrays — ARY_FIND: Find an array in an HDS structure and import it into the ARY system.

ARY_IMPRT: Import an array into the ARY system from HDS.

Inquiring array attributes — ARY_BAD: Determine if an array may contain bad pixels.

ARY_BOUND: Inquire the pixel-index bounds of an array.

ARY_CMPLX: Determine whether an array holds complex values.

ARY_DIM: Inquire the dimension sizes of an array.

ARY_FORM: Obtain the storage form of an array.

ARY_FTYPE: Obtain the full data type of an array.

ARY_ISACC: Determine whether a specified type of array access is available.

ARY_ISMAP: Determine if an array is currently mapped.

ARY_ISBAS: Inquire if an array is a base array.

ARY_ISTMP: Determine if an array is temporary.

ARY_NDIM: Inquire the dimensionality of an array.

ARY_OFFS: Obtain the pixel offset between two arrays.

ARY_SAME: Inquire if two arrays are part of the same base array.

ARY_SIZE: Determine the size of an array.

ARY_STATE: Determine the state of an array (defined or undefined).

ARY_TYPE: Obtain the numeric type of an array.

ARY_VALID: Determine whether an array identifier is valid.

ARY_VERFY: Verify that an array's data structure is correctly constructed.

Creating and deleting arrays — ARY_DELET: Delete an array.

ARY_DUPE: Duplicate an array.

ARY_NEW: Create a new simple array.

ARY_NEWP: Create a new primitive array.

Setting array attributes — ARY_NOACC: Disable a specified type of access to an array.

ARY_RESET: Reset an array to an undefined state.

ARY_SBAD: Set the bad-pixel flag for an array.

ARY_SHIFT: Apply pixel-index shifts to an array.

ARY_STYPE: Set a new type for an array.

Accessing array values — ARY_MAP: Obtain mapped access to an array.

ARY_MAPZ: Obtain complex mapped access to an array.

ARY_UNMAP: Unmap an array.

Creating and controlling identifiers — ARY_ANNUL: Annul an array identifier.

ARY_BASE: Obtain an identifier for a base array.

ARY_CLONE: Clone an array identifier.

ARY_SECT: Create an array section.

ARY_SSECT: Produce a similar array section to an existing one.

ARY_VALID: Determine whether an array identifier is valid.

Creating placeholders — ARY_PLACE: Obtain a placeholder for an array.

ARY_TEMP: Obtain a placeholder for a temporary array.

Copying arrays — ARY_COPY: Copy an array to a new location.

ARY_DUPE: Duplicate an array.

Miscellaneous — ARY_TRACE: Set the internal ARY system error-tracing flag.

21.2.3 REF — References to HDS objects [SUN/31]

REF_ANNUL: Annul a locator which may have been obtained via a reference.

REF_CRPUT: Create a reference object and put a reference in it.

REF_FIND: Obtain a locator to an object (possibly via a reference).

REF_GET: Obtain a locator to a referenced object.

REF_NEW: Create an empty reference object.

REF_PUT: Put a reference into a reference object.

21.2.4 HDS — Hierarchical data system [SUN/92]

In the following lists of routine names, routines which belong to a related set are shown on a single line with 'd' standing for various dimensionalities, and 't' standing for various data types. Not all possible combinations may be available; please refer to SUN/92 for an explicit list.

HDS routines:

HDS_CLOSE: Close container file.

HDS_COPY: Copy object to container file.

HDS_NEW: Create container file.

HDS_ERASE: Erase container file.

HDS_FREE: Free container file.

HDS_GTUNE: Inquire value of tuning parameter.

HDS_LOCK: Lock container file.

HDS_OPEN: Open container file.

HDS_GROUP: Inquire locator group.

HDS_FLUSH: Flush locator group.

HDS_LINK: Link locator group.

HDS_START: Startup locator facility.

HDS_STOP: Rundown locator facility.

HDS_RUN: Run application.

HDS_STATE: Inquire HDS state.

HDS_TUNE: Set HDS parameter.

HDS_SHOW: Show HDS statistics.

HDS_TRACE: Trace object path.

CMP routines:

Inquiry — CMP_LEN: Inquire component precision.

CMP_PRIM: Inquire component primitive.

CMP_SHAPE: Inquire component shape.

CMP_SIZE: Inquire component size.

CMP_STRUC: Inquire component structure.

CMP_TYPE: Inquire component type.

Basic I/O — CMP_GETdt: Read component.

CMP_PUTdt: Write component.

CMP_MAPd: Map component.

CMP_UNMAP: Unmap component.

CMP_MODt: Obtain component.

DAT routines:

Parameter — DAT_ASSOC: Associate a data object with a parameter.

DAT_EXIST: Same as DAT_ASSOC except if there is an error it returns a status value, whereas DAT_ASSOC repeatedly attempts to get a valid locator.

DAT_CANCL: Cancel the association between a data object and a parameter.

DAT_CREAT: Create a data structure component.

DAT_DEF: Suggest values for a parameter.

DAT_DELET: Delete an object associated with a parameter.

DAT_UPDAT: Force HDS update from memory cache.

Inquiry — DAT_NAME: Inquire object name.

DAT_REF: Inquire object reference name.

DAT_PRIM: Inquire object primitive.

DAT_SHAPE: Inquire object shape.

DAT_SIZE: Inquire object size.

DAT_STATE: Inquire object state.

DAT_STRUC: Inquire object structure.

DAT_TYPE: Inquire object type.

DAT_WHERE: Inquire primitive data in file.

DAT_LEN: Inquire primitive precision.

DAT_PREC: Inquire storage precision.

DAT_DREP: Inquire primitive data representation information.

DAT_CLEN: Inquire character string length.

DAT_NCOMP: Inquire number of components.

DAT_THERE: Inquire component existence.

DAT_VALID: Inquire locator valid.

Basic I/O — DAT_GETdt: Read primitive object.

DAT_PUTdt: Write primitive object.

DAT_MAPt: Map primitive object.

DAT_BASIC: Map primitive object as a sequence of bytes.

DAT_UNMAP: Unmap primitive object.

Other (sorted by description) — DAT_ALTER: Alter object size.

DAT_MOULD: Alter object shape.

DAT_ANNUL: Annul locator.

DAT_MSG: Assign object name to message token.

DAT_RETYP: Change object type.

DAT_CLONE: Clone locator.

DAT_COERC: Coerce object shape.

DAT_CCOPY: Copy one structure level.

DAT_COPY: Copy object.

DAT_CCTYP: Create type string.

DAT_NEWdt: Create component.

DAT_TEMP: Create temporary object.

DAT_ERASE: Erase component.

DAT_FIND: Find named component.

DAT_PAREN: Find parent.

DAT_INDEX: Index into component list.

DAT_CELL: Locate cell.

DAT_SLICE: Locate slice.

DAT_MOVE: Move object.

DAT_RENAM: Rename object.

DAT_RESET: Reset object state.

DAT_ERMSG: Translate error status.

DAT_VEC: Vectorise object.

Analysis of data system routine functions:

Data system routine names have the following structure:

```
<pck>_<func><qual>
```

where `<pck>` is the package name, `<func>` represents the function performed by the routine, and `<qual>` is a qualifier which is used to identify different versions of the **GET**, **MAP**, **MOD**, **NEW**, and **PUT** routines. Table 1.1 classifies the routines in terms of their functions, and shows only the `<func>` part of the routine names. The routines shown in the ACTION column perform a function related to the brief description in the FUNCTION column. The routines shown in the QUERY column make inquiries about attributes which are associated with the topic shown in the FUNCTION column. The routines (in the last two columns) are distinguished by type face as follows:

- **XXX** stand for CMP, DAT, and EXC routines.
- *xxx* stand for HDS routines.

	FUNCTION	ACTION	QUERY
SYSTEM CONTROL	Basic control	*start*, *stop*, *run*	*state*
	Tuning	*tune*	*gtune*
CONTAINER FILE	Existence	**TEMP**, *new*, *erase*	*show*
	Access	*open*, *close*	
	Access control	*lock*, *free*	
	I/O	*copy*	
ADDRESSING	Locator group	*link*, *flush*	*group*
	Locator	**CLONE, ANNUL**	*show*, *trace*
		PAREN	**VALID**
	Position		**WHERE**
	Reference	**REF**	
OBJECTS:	General	**RENAM**, **RETYP**	**NAME**, **TYPE**
			PRIM, **STRUC**
			SIZE
			DREP, **CLEN**
structure	Locate component	**FIND**, **INDEX**	**THERE**
	Add component	**NEW**, **MOVE**	**NCOMP**
		COPY, **CCOPY**	
	Modify component	**MOD**	
	Delete component	**ERASE**	
primitive	I/O	**GET**, **PUT**	**LEN**, **PREC**
		BASIC, **MAP**, **UNMAP**	
	State	**RESET**	**STATE**
array	Subset of elements	**SLICE**, **CELL**	**SHAPE**
	Re-mapping	**VEC**, **COERC**	
	Change size	**ALTER**, **MOULD**	
MISCELLANEOUS	String manipulation	**CCTYP**, **MSG**	
	Error handling	**ERMSG**	

Table 1.1: Analysis of Data System routine functions.

21.3 Message and Error systems

21.3.1 MSG/ERR — Message and Error reporting [SUN/104]

MSG routines:

MSG_BLANK: Output a blank line.
MSG_FMTt: Assign a value to a message token (formatted).
MSG_IFGET: Get the filter level from parameter system.
MSG_IFLEV: Return the current filter level for conditional message output.
MSG_IFSET: Set the filter level for conditional message output.
MSG_LOAD: Expand and return a message.
MSG_OUT: Output a message.
MSG_OUTIF: Conditionally deliver the text of a message to the user.
MSG_RENEW: Renew any annulled message tokens in the current context.
MSG_SETt: Assign a value to a message token (concise).

ERR routines:

ERR_ANNUL: Annul the contents of the current error context.
ERR_BEGIN: Begin a new error reporting environment.
ERR_END: End the current error reporting environment.
ERR_FIOER: Assign a Fortran I/O error message to a token.
ERR_FLUSH: Flush the current error context.
ERR_LEVEL: Inquire the current error context level.
ERR_LOAD: Return error messages from the current error context.
ERR_MARK: Mark (start) a new error context.
ERR_REP: Report an error message.
ERR_RLSE: Release (end) the current error context.
ERR_STAT: Inquire the last reported error status.
ERR_SYSER: Assign an operating system error message to a token.

EMS routines:

The EMS routines will not be listed here as they are intended for internal development work rather than for writing application programs.

21.4 Graphics system

21.4.1 NCAR/SNX — Graphics utilities [SUN/88, SUN/90]

Currently there are no NCAR 'parameter routines'. A full list of 'stand-alone' NCAR routine names is given in SUN/88. The routines are implemented within groupings called *utilities* which have the following names:

AUTOGRAPH: Draw and annotate curves or families of curves.

CONRAN: Contour irregularly spaced data labelling the contour lines.

CONRAQ: Like CONRAN, but faster because it has no labelling capability.

CONRAS: Like CONRAN, but slower because lines are smoothed and crowded lines are removed.

CONREC: Contour 2-d arrays, labelling the contour lines.

CONRECQCK: Like CONREC, but faster because it has no labelling capability.

CONRECSUPR: Like CONREC, but slower because lines are smoothed and crowded lines are removed.

DASHCHAR: Software dashed line package with labelling capability.

DASHLINE: Like DASHCHAR, but faster because it has no labelling capability.

DASHSMTH: Like DASHCHAR, but slower because lines are smoothed.

DASHSUPR: Like DASHCHAR, but slower because lines are smoothed and crowded lines are removed.

EZMAP: Plot continental, national, US state boundaries in one of nine map projections.

GRIDAL: Draw graph paper, axis etc.

HAFTON: Draw halftone (greyscale) picture from a 2-d array.

HSTGRM: Plot histograms.

ISOSRF: Plot iso-value surfaces (with hidden lines removed) from a 3-d array.

ISOSRFHR: Plot iso-value surfaces (with hidden lines removed) from a high resolution 3-d array.

PWRITX: Plot high quality software characters.

PWRITY: Plot simple software characters.

PWRZI: Draw characters in three-space, for use with the ISOSRF utility.

PWRZS: Draw characters in three-space, for use with the SRFACE utility.

PWRZT: Draw characters in three-space, for use with the THREED utility.

SRFACE: Plot a 3-d display of a surface (with hidden lines removed) from a 2-d array.

STRMLN: Plot a representation of a vector flow of any field for which planar vector components are given on a regular rectangular lattice.

THREED: Provide a three space line drawing capability.

VELVCT: Draw a 2-d velocity field by drawing arrows from the data locations.

SNX routines:

Functions — SNX_AGGUX: Transform grid X coordinate to user X coordinate.

SNX_AGGUY: Transform grid Y coordinate to user Y coordinate.

SNX_AGUGX: Transform user X coordinate to grid X coordinate.

SNX_AGUGY: Transform user Y coordinate to grid Y coordinate.

Subroutines — SNX_AGOP: Make AUTOGRAPH/SGS/GKS ready for plotting.

SNX_EZRXY: Plot x,y data, with labelling, using AUTOGRAPH.

SNX_AGLAB: Set up an AUTOGRAPH label.

SNX_CHSET: Select one of the two NCAR Roman fonts.

SNX_AGSAV: Save the state of AUTOGRAPH.

SNX_AGRES: Restore the state of AUTOGRAPH.

SNX_TO: Switch between NCAR and SGS plotting.

SNX_AGWV: Make the AUTOGRAPH graph window match the current viewport (= zone if using SGS).

SNX_AGCS: Make the current SGS zone world coordinates match the AUTOGRAPH grid coordinate system.

SNX_CURS: Read a cursor position.

SNX_WRTST: Plot a character string.

21.4.2 PGPLOT — Graphics library [SUN/15]

Control — PGADVANCE: See PGPAGE.

PGASK: Control new page prompting.

PGBBUF: Begin batch of output (buffer).

PGBEGIN: Begin PGPLOT, open output device.

PGEBUF: End batch of output (buffer).

PGEND: Terminate PGPLOT.

PGPAGE: Advance to a new page or clear screen.

PGPAPER: Change the size of the view surface.

PGUPDT: Update display.

Windows and viewports — PGBOX: Draw labeled frame around viewport.

PGENV: Set window and viewport and draw labeled frame.

PGVPORT: Set viewport (normalized device coordinates).

PGVSIZE: Set viewport (inches).

PGVSTAND: Set standard (default) viewport.

PGWINDOW: Set window.

PGWNAD: Set window and adjust viewport to same aspect ratio.

Primitive drawing — PGDRAW: Draw a line from the current pen position to a point.

PGLINE: Draw a polyline.

PGMOVE: Move pen.

PGPOINT: Draw one or more graph markers.

PGPOLY: Fill a polygonal area with shading.

PGRECT: Draw a rectangle, using fill-area attributes.

Text — PGLABEL: Write labels for x-axis, y-axis, and top of plot.

PGMTEXT: Write text at position relative to viewport.

PGPTEXT: Write text at arbitrary position and angle.

PGTEXT: Write text (horizontal, left-justified).

Attribute setting — PGSCF: Set character font.

PGSCH: Set character height.

PGSCI: Set color index.

PGSCR: Set color representation.

PGSFS: Set fill-area style.

PGSHLS: Set color representation using HLS system.

PGSLS: Set line style.

PGSLW: Set line width.

Higher-level drawing — PGBIN: Histogram of binned data.

PGCONS: Contour map of a 2-d data array (fast algorithm).

PGCONT: Contour map of a 2-d data array (contour-following).

PGCONX: Contour map of a 2-d data array (non-rectangular).

PGERRX: Horizontal error bar.

PGERRY: Vertical error bar.

PGFUNT: Function defined by X = F(T), Y = G(T).

PGFUNX: Function defined by Y = F(X).

PGFUNY: Function defined by X = F(Y).

PGGRAY: Gray-scale map of a 2-d data array.

PGHI2D: Cross-sections through a 2-d data array.

PGHIST: Histogram of unbinned data.

Interactive graphics (cursor) — PGCURSE: Read cursor position.

PGLCUR: Draw a line using the cursor.

PGNCURSE: Mark a set of points using the cursor.

PGOLIN: Mark a set of points using the cursor.

Inquiry — PGQCF: Inquire character font.

PGQCH: Inquire character height.

PGQCI: Inquire color index.

PGQCR: Inquire color representation.

PGQFS: Inquire fill-area style.

PGQINF: Inquire general information.

PGQLS: Inquire line style.

PGQLW: Inquire line width.

PGQVP: Inquire viewport size and position.

PGQWIN: Inquire window boundary coordinates.

Utilities — PGETXT: Erase text from graphics display.

PGIDEN: Write username, date, and time at bottom of plot.

PGLDEV: List available device types.

PGNUMB: Convert a number into a plottable character string.

PGRND: Find the smallest “round” number greater than x.

PGRNGE: Choose axis limits.

21.4.3 NAG — Graphics library [SUN/29]

Axes, Grids, Borders, and Titles — J06AAF: Pair of axes for current data region, automatic annotation.

J06ABF: Pair of axes for current data region, user-specified annotation.

J06ACF: Grid for the current data region, automatic annotation.

J06ADF: Grid for the current data region, user-specifiable annotation.

J06AEF: Scaled border for current data region, automatic annotation.

J06AFF: Scaled border for current data region, user-specified annotation.

J06AGF: Single axis under user control.

J06AHF: Plot title, centred at the top of the current data region.

J06AJF: Axis title, centred at side or bottom of current data region.

J06AKF: Pair of logarithmic axes for current data region, automatic annotation.

J06ALF: Logarithmic grid for current data region, automatic annotation.

J06AMF: Logarithmic border for current data region, automatic annotation.

J06ANF: Single logarithmic axis under user control.

J06APF: Single axis with user-specified position of tick marks and annotation.

J06AQF: Single axis with user-supplied annotation, and control over position of tick marks and annotation.

Point plotting and Straight line drawing — J06BAF: Plot data points with optional straight lines and markers.

J06BBF: Linear regression line using output from G02CAF, G02CBF, G02CCF or G02CDF.

J06BCF: Plot a series of data points with optional error bars.

J06BYF: Key to current NAG pen styles.

J06BZF: Key to current line styles and/or markers.

Curve drawing — J06CAF: Plot single-valued curve through data points.

J06CBF: Plot single-valued curve through data points, called point-wise.

J06CCF: Plot possibly multi-valued curve through data points.

J06CDF: Plot possibly multi-valued curve through data points, called point-wise.

ODE graphics — J06DAF: Plot a graph of components of the solution of a system of ODEs, comprehensive version, using option setting.

General function drawing — J06EAF: Plot user-supplied function over specified range.

J06EBF: Plot user-supplied function over specified range, called point-wise.

J06ECF: Plot a parametric curve.

Special function drawing — J06FAF: Plot cubic spline in an interval, from its B-spline representation.

J06FBF: Plot polynomial represented in Chebyshev form, using output from E02ADF or E02AFF.

Contouring — J06GAF: Easy-to-use contour map, data on regular rectangular grid.

J06GBF: Comprehensive contour map, data on regular rectangular grid.

J06GCF: Easy-to-use contour map, data on irregular rectangular grid.

J06GDF: Comprehensive contour map, data on irregular rectangular grid.

J06GEF: Easy-to-use contour map, user-supplied function.

J06GFF: Comprehensive contour map, user-supplied function.

J06GGF: Comprehensive contour map, using option setting, data scattered.

J06GYF: Key to contour plot produced with J06GGF.

J06GZF: Key to contour indices.

Surface viewing — J06HAF: Easy-to-use isometric surface view, data on regular rectangular grid.

J06HBF: Comprehensive isometric surface view, data on regular rectangular grid.

J06HCF: Perspective surface view, data on regular rectangular grid.

J06HDF: Easy-to-use perspective surface view, data on irregular rectangular grid.

J06HEF: Comprehensive perspective surface view, data on irregular rectangular grid.

J06HFF: Perspective surface view, sections parallel to either horizontal axis.

J06HGF: Perspective view of a 3-d histogram.

J06HHF: Annotate data points on a perspective surface view.

J06HJF: Annotate individual data blocks on a 3-d histogram.

J06HKF: Isometric surface view of a 2-d function f(x,y).

J06HLF: Perspective surface view of a 2-d function f(x,y).

Data presentation — J06JAF: Shaded bar chart, vertical, various styles (easy-to-use).

J06JBF: Shaded bar chart, vertical, various styles (comprehensive).

J06JCF: Shaded bar chart, variable width bars.

J06JDF: Shaded bar chart, horizontal, various styles (easy-to-use).

J06JEF: Shaded bar chart, horizontal, various styles (comprehensive).

J06JFF: Shaded block chart, variable width and height blocks.

J06JGF: Marker diagram, sorts data into bins.

J06JHF: Marker diagram, subdivision of bins.

J06JKF: Shaded pie chart, (easy-to-use).

J06JLF: Shaded pie chart, (comprehensive).

J06JMF: Shaded bar chart, vertical or horizontal, no subdivision of bins (easy-to-use).

J06JNF: Frequency distribution diagram, various styles, integer grid input.

J06JPF: Frequency distribution diagram and scatter plot, various styles, x,y input data.

J06JYF: Key to numeric values.

J06JZF: Key to area fill styles.

Vector field plotting — J06KAF: Plot a 2-d vector field diagram.

J06KBF: Plot a 3-d vector field diagram.

J06KCF: Plot a 2-d vector field diagram from a user-supplied function.

J06KDF: Plot a 3-d vector field diagram from a user-supplied function.

Statistical graphics — J06SAF: Plot a time-series, and optionally user-supplied forecasts and standard error limits.

J06SBF: Plot autocorrelation or partial autocorrelation function of a time series.

J06SCF: Plot a cumulative normal probability graph. Data may be grouped together and a histogram plotted.

J06SDF: Plot a linear regression line and optional confidence limits for data and line.

J06SEF: Draw box and whisker plots.

I/O utilities — J06VAF: Return or define the unit number for error messages.

J06VBF: Return or define the unit number for advisory messages.

J06VCF: Return or define the unit number for command sequence output.

J06VDF: Return or define the unit number from which Hershey fonts will be read (used only by PC Graphics implementations).

J06VEF: Return or define the unit number to which SAVE options should be directed.

J06VFF: Read a set of optional parameters from an external file.

J06VGF: Supply individual optional parameter for a specified routine.

21.4.4 SGS — Simple graphics system [SUN/85]

Parameter — SGS_ASSOC: Associate an SGS zone with a parameter.

SGS_CANCL: Cancel the association of an SGS zone with a parameter.

SGS_ANNUL: Release an SGS zone and annul its descriptor.

Control — SGS_OPEN: Open SGS & GKS, opening and activating one workstation.

SGS_INIT: Open SGS & GKS, without opening a workstation.

SGS_CLOSE: Shut down SGS & GKS.

SGS_OPNWK: Open a new workstation and select it.

SGS_CLSWK: Close a workstation.

SGS_WLIST: Output a list of available workstations.

SGS_SPEN: Select a pen for line and text drawing.

SGS_IPEN: Inquire current pen number.

SGS_FLUSH: Complete all pending output.

SGS_ISLER: Inquire whether device has selective erase capability.

SGS_CLRFG: Set or clear the 'clear display on open' flag.

Zones — SGS_ZONE: Create and select a zone of the specified extent.

SGS_ZSHAP: Create and select a zone of the specified aspect ratio.

SGS_ZSIZE: Create and select a zone of the specified size.

SGS_ZPART: Partition current zone into NX by NY pieces.

SGS_SW: Set window of current zone to given bounds.

SGS_SELZ: Select another zone.

SGS_RELZ: Release the specified zone.

SGS_CLRZ: Clear the current zone (even if this means clearing the whole display surface).

SGS_ICURZ: Inquire ID of current zone.

SGS_IZONE: Inquire window bounds and size on the display surface for the current zone.

SGS_IDUN: Inquire the plotting resolution of the current zone.

SGS_TPZ: Transform position from one zone to another.

SGS_BZNDC: Set the NDC extent of a base zone.

Plotting lines — SGS_BPOLY: Begin a new polyline.

SGS_APOLY: Append a new line to the polyline.

SGS_OPOLY: Output the polyline.

SGS_IPLXY: Inquire end of current polyline.

SGS_LINE: Begin a new polyline with a single line.

SGS_BOX: Draw a rectangle.

SGS_CIRCL: Draw a circle.

SGS_ARC: Draw an arc of a circle.

SGS_CLRBL: Clear a rectangular area (if this can be done without affecting the rest of the display surface).

Plotting text — SGS_BTEXT: Begin a new text string.

SGS_ATEXT: Append text to the current string.

SGS_OTEXT: Output the text string.

SGS_ATXI: Format an integer onto the current text string.

SGS_ATXR: Format a real number onto the current text string.

SGS_ATXL: Append text to the current text string, omitting trailing blanks.

SGS_ATXB: Append right justified field to the current text string.

SGS_TX: Begin a new text string with a string.

SGS_TXI: Begin a new text string with a formatted integer.

SGS_TXR: Begin a new text string with a formatted real number.

SGS_ITXB: Inquire status of current text string.

Plotting markers — **SGS_MARK:** Draw a single marker.

SGS_MARKL: Draw a marker at current end of polyline.

Attributes of characters — **SGS_SFONT:** Select text font.

SGS_SPREC: Specify text precision.

SGS_SHTX: Specify character height.

SGS_SARTX: Specify text aspect ratio.

SGS_SUPTX: Specify character orientation.

SGS_SSPTX: Specify character spacing.

SGS_STXJ: Specify text alignment.

SGS_ITXA: Inquire text attributes.

Input — **SGS_CUVIS:** Set visibility of cursor.

SGS_DEFCH: Define valid choice keys.

SGS_DISCU: Disable sample mode for cursor.

SGS_ENSCU: Enable sample mode for cursor.

SGS_ICUAV: Inquire cursor availability.

SGS_INCHO: Inquire number of choices.

SGS_REQCH: Request choice.

SGS_REQCU: Request cursor position.

SGS_SAMCU: Sample cursor position.

SGS_SELCH: Select choice device.

SGS_SETCU: Set cursor position.

GKS Inquires — **SGS_ICURW:** Inquire workstation ID for the current zone.

SGS_WIDEN: Translate workstation name to GKS type and connection ID.

SGS_WNAME: Get list of workstation names.

21.4.5 GKS — Graphical kernel system [SUN/83]

The GKS routines should not be used in ADAM programs, except in special cases, so they will not be listed here. A full list of the stand-alone routines is available in the RAL GKS Guide, in addition to which there exist the following 'parameter routines':

GKS_ASSOC: Associate a GKS workstation with a parameter and open it.

GKS_CANCL: Cancel the association of a GKS workstation with a parameter.

GKS_ANNUL: Release a GKS workstation and annul its descriptor.

GKS_GSTAT: Inquire if GKS has reported an error.

GKS_DEACT: De-activate GKS.

21.4.6 IDI — Image display interface [SUN/65]

Parameter — IDI_ASSOC: Associate a display device with a parameter.

IDI_CANCL: Cancel the association of a display device with a parameter.

IDI_ANNUL: Release a display device and annul its descriptor.

Control — IIDOPN: Open display.

IIDCLO: Close display.

IIDRST: Reset display.

IIDUPD: Update display.

IIDERR: Get error.

Configuration — IIDQDV: Query device characteristics.

IIDQCI: Query capabilities integer.

IIDQCR: Query capabilities real.

IIDQDC: Query defined configuration.

IIDSEL: Select configuration.

Memories — IIMSMV: Set memory visibility.

IIZWSC: Write memory scroll.

IIZWZM: Write memory zoom.

IIZRSZ: Read memory scroll and zoom.

IIMSLT: Select memory look-up tables.

IIMSDP: Select display path.

IIMWMY: Write memory.

IIMCMY: Clear memory.

IIMRMY: Read memory.

IIMSTW: Set transfer window.

Graphics — IIGPLY: Draw polyline.

IIGTXT: Plot text.

Look-up table — IILWIT: Write intensity transformation table.

IILRIT: Read intensity transformation table.

IILWLT: Write video look-up tables.

IILRLT: Read video look-up tables.

Zoom and Pan — IIZWZP: Write display zoom and pan.

IIZRZP: Read display zoom and pan.

Cursor — IICINC: Initialize cursor.

IICSCV: Set cursor visibility.

IICRCP: Read cursor position.

IICWCP: Write cursor position.

Region of interest — IIRINR: Initialize rectangular region of interest.

IIRSRV: Set visibility rectangular region of interest.

IIRRRI: Read rectangular region of interest.

IIRWRI: Write rectangular region of interest.

Interaction — IIIENI: Enable interaction.

IIIEIW: Execute interaction and wait.

IIISTI: Stop interactive input.

IIIQID: Query interactor description.

IIIGLD: Get locator displacement.

IIIGIE: Get integer evaluator.

IIIGRE: Get real evaluator.

IIIGLE: Get logical evaluator.

IIIGSE: Get string evaluator.

Miscellaneous — IIDSNP: Create snapshot.

IIMBLM: Blink memories.

IIDSSS: Set split screen.

IILSBV: Set intensity bar visibility.

IIDIAG: Diagnostic routine.

IIEGEP: Get escape parameter.

IIEPEP: Put escape parameter.

Workstation interface — IIDENC: Enable configuration.

IIDAMY: Allocate memory.

IIDSTC: Stop configuration.

IIDRLC: Release configuration.

Additional — IDI_CLRFG: Set clear flag.

21.4.7 AGI — Applications graphics interface [SUN/48]

Parameter — AGI_ASSOC: Associate a workstation with a parameter.
AGI_CANCL: Cancel parameter association.
AGI_ANNUL: Annul the picture identifier.

Control — AGI_BEGIN: Mark the beginning of a new AGI scope.
AGI_CLOSE: Close AGI in a non-ADAM environment.
AGI_END: Mark the end of an AGI scope.
AGI_NUPIC: Create a new picture in the database.
AGI_OPEN: Open AGI in a non-ADAM environment.
AGI_PDEL: Delete all pictures on a workstation.
AGI_SELP: Select the given picture as the current one.
AGI_SROOT: Select the root picture for searching.

Inquiries — AGI_IBASE: Inquire base picture for current workstation.
AGI_ICOM: Inquire comment for current picture.
AGI_ICURP: Inquire current picture identifier.
AGI_INAME: Inquire name of current picture.
AGI_IPOBS: Inquire if current picture obscured by another.
AGI_ISAMD: Inquire if pictures are on same device.
AGI_ITOBS: Inquire if test points are obscured.
AGI_IWOCO: Inquire world coordinates of current picture.

Recall — AGI_RCF: Recall first picture of specified name.
AGI_RCFP: Recall first picture embracing a position.
AGI_RCL: Recall last picture of specified name.
AGI_RCLP: Recall last picture embracing a position.
AGI_RCP: Recall preceding picture of specified name.
AGI_RCPP: Recall preceding picture embracing a position.
AGI_RCS: Recall succeeding picture of specified name.
AGI_RCSP: Recall succeeding picture embracing a position.

Labels — AGI_ILAB: Inquire label of a picture.
AGI_SLAB: Store label in picture.

Reference — AGI_GTREF: Get a reference to an HDS object.
AGI_PTREF: Write a reference to an HDS object.

Transformations — AGI_TDDTW: Transform double precision data to world coordinates.
AGI_TDTOW: Transform data to world coordinates.
AGI_TNEW: Store a transformation in the database.
AGI_TWTDD: Transform double precision world to data coordinates.
AGI_TWTOD: Transform world to data coordinates.

Interface to IDI — AGD_ACTIV: Initialise IDI.
AGD_DEACT: Close down IDI.
AGD_NWIND: Define an IDI window from the current picture.
AGD_SWIND: Save an IDI window in the database.

Interface to PGPLOT — AGP_ACTIV: Initialise PGPLOT.
AGP_DEACT: Close down PGPLOT.
AGP_NVIEW: Create a new viewport from the current picture.
AGP_SVIEW: Save the current viewport in the database.

Interface to SGS — AGS_ACTIV: Initialise SGS.
AGS_DEACT: Close down SGS.
AGS_NZONE: Create a new zone from the current picture.
AGS_SZONE: Save the current zone in the database.

21.4.8 GNS — Graphics workstation name service [SUN/57]

Get names — **GNS_GTN:** Get terminal name.

GNS_GWNG: Get next GKS workstation name.

GNS_GWNI: Get next IDI workstation name.

Inquiries — **GNS_IANG:** Inquire AGI name of GKS workstation.

GNS_IANI: Inquire AGI name of IDI workstation.

GNS_IDNG: Inquire device name of GKS workstation.

GNS_IETG: Inquire string to erase text screen.

GNS_IGAG: Inquire GKS workstation name from AGI name.

GNS_IIAI: Inquire IDI workstation name from AGI name.

GNS_ITWCG: Inquire a workstation characteristic from its type.

GNS_IWCG: Inquire a workstation characteristic.

GNS_IWSG: Inquire workstation scale.

Control — **GNS_START:** Start the GNS system for the specified package.

GNS_STOP: Stop the GNS system for the specified package.

Translation — **GNS_TNDG:** Translate a name and device to a GKS device specification.

GNS_TNG: Translate a name to a GKS device specification.

GNS_TNI: Translate a name to a IDI device specification.

21.5 Input/output systems

21.5.1 FIO/RIO — File I/O [SUN/143]

FIO routines:

Parameter — FIO_ASSOC: Associate a parameter with a sequential file.
  FIO_CANCL: Cancel the association between a parameter and a sequential file.
File management and positioning — FIO_OPEN: Create and open a sequential file.
  FIO_CLOSE: Close a sequential file.
  FIO_RWIND: Rewind a sequential file.
  FIO_ERASE: Delete a sequential file.
  FIO_SERR: Set error status.
  FIO_DEACT: Deactivate FIO.
  FIO_STOP: Close down FIO.
File information and Fortran unit numbers — FIO_FNAME: Get a file's full name.
  FIO_UNIT: Get a file's Fortran unit number.
  FIO_GUNIT: Get an unused Fortran unit number.
  FIO_PUNIT: Release a Fortran unit number.
I/O — FIO_READ: Read a sequential record.
  FIO_READF: Fast read a sequential record.
  FIO_WRITE: Write a sequential record.

RIO routines:

Parameter — RIO_ASSOC: Associate a parameter with a direct-access file.
  RIO_CANCL: Cancel the association between a direct-access file and a parameter.
File management — RIO_OPEN: Open a direct-access file.
  RIO_ERASE: Delete a direct-access file.
I/O — RIO_READ: Read a direct-access record.
  RIO_WRITE: Write a direct-access record.

21.5.2 MAG — Magnetic tape I/O [APN/1]

Parameter — MAG_ASSOC: Associate a parameter with a tape drive.

MAG_CANCL: Cancel the association between a parameter and a tape drive.

MAG_ANNUL: Annul a tape descriptor.

Device management — MAG_ALOC: Allocate a tape drive.

MAG_DEAL: De-allocate a tape drive.

MAG_MOUNT: Mount a tape on a drive.

MAG_DISM: Dismount a tape on a drive.

Tape positioning — MAG_JUMP: Skip blocks.

MAG_SKIP: Skip files.

MAG_JEOV: Skip past an EOV marker.

MAG_REW: Position a tape at its beginning.

MAG_MOVE: Position a tape at an absolute position.

MAG_POS: Obtain an absolute position.

MAG_SET: Set an absolute position.

I/O — MAG_READ: Read a block from a tape.

MAG_WRITE: Write a block to a tape.

MAG_WTM: Write a tape mark to a tape.

21.6 Database system

21.6.1 CHI — Catalogue handling [SUN/119]

Control — CHI_OPEN: Activate the interface.

CHI_CLOSE: Deactivate the interface.

List catalogues — CHI_AVAILCATS: Get a list of currently available catalogues.

Manipulate data — CHI_GETNUMENTS: Get the number of entries in a catalogue.

CHI_GETP: Get the parameter names and the number of parameters.

CHI_GETF: Get the field names and the number of fields.

CHI_GETFINF: Get information associated with a field.

CHI_GETPINF: Get information associated with a parameter.

CHI_ADDP: Add a parameter to a catalogue.

CHI_DELP: Delete a parameter from a catalogue.

Create catalogues — CHI_NOENT: Create a new catalogue with no entries.

CHI_CREATDUP: Create a duplicate catalogue with no entries.

CHI_SELFLDS: Create a new catalogue with fields selected from an existing catalogue.

CHI_SEARCH: Create a new catalogue with entries selected from an existing catalogue.

CHI_REJECT: Similar to CHI_SEARCH but with entries that *fail* selection criteria.

CHI_JOIN: Create a new catalogue by joining two existing catalogues based on join criteria.

CHI_MERGE: Create a new catalogue by merging two existing catalogues.

CHI_NEWFLD: Create a new catalogue by adding a new field to an existing catalogue.

Delete catalogues — CHI_DELCAT: Delete a catalogue.

Put information into catalogues — CHI_PUTCENT: Put a complete entry into a catalogue.

CHI_PUTENT: Put a set of values that constitute an entry into a catalogue.

Get information out of catalogues — CHI_GETCENT: Get a complete entry from a catalogue.

CHI_GETCENTPOS: Get a complete entry from a given position in a catalogue.

CHI_GETVAL: Get a field value(s) from the next entry in a catalogue.

CHI_GETSVALS: Get a field value(s) for a specified number of entries in a catalogue.

21.7 Utilities

The following libraries are not specific to ADAM but, with the increased interest in writing portable code, they can be of considerable use to the writer of ADAM applications.

21.7.1 CHR — Character handling [SUN/40]

Decoding — **CHR_CTOD:** Read a double precision number from a character string.

CHR_CTOI: Read an integer number from a character string.

CHR_CTOL: Read a logical value from a character string.

CHR_CTOR: Read a real number from a character string.

CHR_DCWRD: Returns all the words in a string.

CHR_HTOI: Read an integer from a hex string.

CHR_OTOI: Read an integer from an octal string.

Encoding and Formatting — **CHR_CTOC:** Write a character value into a string.

CHR_DTOC: Encode a double precision value as a string.

CHR_ITOC: Encode an integer value as a string.

CHR_LTOC: Encode a logical value as a string.

CHR_RTOC: Encode a real value as a string.

CHR_PUTC: Copy one string into another at given position.

CHR_PUTD: Put double precision value into string at given position.

CHR_PUTI: Put integer value into string at given position.

CHR_PUTL: Put logical value into string at given position

CHR_PUTR: Put real value into string at given position.

CHR_RTOAN: Write a real into character string as hr/deg:min:sec.

Inquiry — **CHR_DELIM:** Locate substring with given delimiter character.

CHR_EQUAL: Determine whether two strings are equal.

CHR_FANDL: Find the indices of the first and last non-blank characters.

CHR_FIWE: Find next end of word.

CHR_FIWS: Find start of next word.

CHR_INDEX: Find the index of a substring in a string.

CHR_INSET: Determine whether a string is a member of a set.

CHR_ISALF: Determine whether a character is alphabetic.

CHR_ISALM: Determine whether a character is alphanumeric.

CHR_ISDIG: Determine whether a character is a digit.

CHR_ISNAM: Determine whether a string is a valid name.

CHR_LEN: Find used length of string.

CHR_SIMLR: Determine whether two strings are equal apart from case.

CHR_SIZE: Find the declared size of string.

String manipulation — **CHR_APPND:** Copy one string into another (ignoring trailing blanks).

CHR_CLEAN: Remove all non-printable ASCII characters from a string.

CHR_COPY: Copy one string to another, checking for truncation.

CHR_FILL: Fill a string with a given character.

CHR_LCASE: Convert a string to lower case.

CHR_LDBLK: Remove leading blanks from a string.

CHR_LOWER: Give lower case equivalent of a character.

CHR_MOVE: Move one string into another (ignoring trailing blanks).

CHR_RMBLK: Remove all blanks from a string in situ.

CHR_SWAP: Swap two single-character variables.

CHR_TERM: Terminate string by padding out with blanks.

CHR_TRUNC: Truncate string rightwards from a given delimiter.

CHR_UCASE: Convert a string to upper case.

CHR_UPPER: Give upper case equivalent of a character.

21.7.2 CNF/F77 — Mixed language programming [SGP/5]

CNF functions:

CNF is a set of C functions to handle the difference between Fortran and C character strings.

Import a Fortran string to C — CNF_CREIB: Create a temporary C string and import a blank filled Fortran string into it.

CNF_CREIM: Create a temporary C string and import a Fortran string into it.

CNF_IMPB: Import a Fortran string into a C string, retaining trailing blanks.

CNF_IMPBN: Import no more than max characters from a Fortran string into a C string, retaining trailing blanks.

CNF_IMPN: Import no more than max characters from a Fortran string into a C string.

CNF_IMPRT: Import a Fortran string into a C string.

Export a C string to Fortran — CNF_EXPN: Export a C string to a Fortran string, copying given a maximum number of characters.

CNF_EXPRT: Export a C string to a Fortran string.

String lengths — CNF_LENC: Find the length of a C string.

CNF_LENF: Find the length of a Fortran string.

Miscellaneous — CNF_COPYF: Copy one Fortran string to another Fortran string.

CNF_CREAT: Create a temporary C string and return a pointer to it.

CNF_FREE: Return temporary space.

F77 macros:

F77 is a set of C macros to handle Fortran to C subroutine linkage.

Declaration of a C function — F77_BYTE_FUNCTION: Declare a function that returns a BYTE value.

F77_CHARACTER_FUNCTION: Declare a function that returns a CHARACTER value.

F77_DOUBLE_FUNCTION: Declare a function that returns a DOUBLE PRECISION value.

F77_INTEGER_FUNCTION: Declare a function that returns an INTEGER value.

F77_LOGICAL_FUNCTION: Declare a function that returns a LOGICAL value.

F77_REAL_FUNCTION: Declare a function that returns a REAL value.

F77_SUBROUTINE: Declare a SUBROUTINE.

F77_WORD_FUNCTION: Declare a function that returns a WORD value.

Arguments of a C function — BYTE: Declare a BYTE argument.

BYTE_ARRAY: Declare a BYTE array argument.

CHARACTER: Declare a CHARACTER argument.

CHARACTER_ARRAY: Declare a CHARACTER array argument.

CHARACTER_RETURN_VALUE: Declare an argument that will be the return value of a CHARACTER function.

DOUBLE: Declare a DOUBLE PRECISION argument.

DOUBLE_ARRAY: Declare a DOUBLE PRECISION array argument.

INTEGER: Declare an INTEGER argument.

INTEGER_ARRAY: Declare an INTEGER array argument.

LOGICAL: Declare a LOGICAL argument.

LOGICAL_ARRAY: Declare a LOGICAL array argument.

POINTER: Declare a POINTER argument.

REAL: Declare a REAL argument.

REAL_ARRAY: Declare a REAL array argument.

TRAIL: Declare hidden trailing arguments.

TRAIL_ARG: Pass the length of a CHARACTER argument to a Fortran routine.

WORD: Declare a WORD argument.

WORD_ARRAY: Declare a WORD array argument.

Generate pointers to arguments — GENPTR_BYTE: Generate a pointer to a BYTE argument.

GENPTR_BYTE_ARRAY: Generate a pointer to a BYTE array argument.

GENPTR_CHARACTER: Generate a pointer to a CHARACTER argument.

GENPTR_CHARACTER_ARRAY: Generate a pointer to a CHARACTER array argument.

GENPTR_DOUBLE: Generate a pointer to a DOUBLE PRECISION argument.

GENPTR_DOUBLE_ARRAY: Generate a pointer to a DOUBLE PRECISION array argument.

GENPTR_INTEGER: Generate a pointer to an INTEGER argument.

GENPTR_INTEGER_ARRAY: Generate a pointer to an INTEGER array argument.

GENPTR_LOGICAL: Generate a pointer to a LOGICAL argument.

GENPTR_LOGICAL_ARRAY: Generate a pointer to a LOGICAL array argument.

GENPTR_POINTER: Generate a pointer to a POINTER argument.

GENPTR_REAL: Generate a pointer to a REAL argument.

GENPTR_REAL_ARRAY: Generate a pointer to a REAL array argument.

GENPTR_WORD: Generate a pointer to a WORD argument.

GENPTR_WORD_ARRAY: Generate a pointer to a WORD array argument.

Data types — F77_BYTE_TYPE: Define the type BYTE.

F77_CHARACTER_TYPE: Define the type CHARACTER.

F77_DOUBLE_TYPE: Define the type DOUBLE PRECISION.

F77_INTEGER_TYPE: Define the type INTEGER.

F77_LOGICAL_TYPE: Define the type LOGICAL.

F77_REAL_TYPE: Define the type REAL.

F77_WORD_TYPE: Define the type WORD.

Logical values — F77_FALSE: The logical value FALSE.

F77_ISFALSE: Is this the Fortran logical value false?

F77_ISTRUE: Is this the Fortran logical value true?

F77_TRUE: The logical value TRUE.

External names — F77_EXTERNAL_NAME: The external name of a function.

Common blocks — F77_BLANK_COMMON: Refer to blank common.

F77_NAMED_COMMON: Refer to a named common block.

Declaring variables for passing to a Fortran routine — DECLARE_BYTE: Declare a BYTE variable.

DECLARE_BYTE_ARRAY: Declare a BYTE array.

DECLARE_CHARACTER: Declare a CHARACTER variable.

DECLARE_CHARACTER_ARRAY: Declare a CHARACTER array.

DECLARE_DOUBLE: Declare a DOUBLE variable.

DECLARE_DOUBLE_ARRAY: Declare a DOUBLE array.

DECLARE_INTEGER: Declare an INTEGER variable.

DECLARE_INTEGER_ARRAY: Declare an INTEGER array.

DECLARE_LOGICAL: Declare a LOGICAL variable.

DECLARE_LOGICAL_ARRAY: Declare a LOGICAL array.

DECLARE_REAL: Declare a REAL variable.

DECLARE_REAL_ARRAY: Declare a REAL array.

DECLARE_WORD: Declare a WORD variable.

DECLARE_WORD_ARRAY: Declare a WORD array.

Passing arguments to a Fortran routine — BYTE_ARG: Pass a BYTE argument to a Fortran routine.

BYTE_ARRAY_ARG: Pass a BYTE array argument to a Fortran routine.

CHARACTER_ARG: Pass a CHARACTER argument to a Fortran routine.

CHARACTER_ARRAY_ARG: Pass a CHARACTER array argument to a Fortran routine.

DOUBLE_ARG: Pass a DOUBLE argument to a Fortran routine.

DOUBLE_ARRAY_ARG: Pass an actual DOUBLE array argument to a Fortran routine.

F77_CALL: Call a Fortran routine from C.

INTEGER_ARG: Pass an INTEGER argument to a Fortran routine.

INTEGER_ARRAY_ARG: Pass an INTEGER array argument to a Fortran routine.

LOGICAL_ARG: Pass a LOGICAL argument to a Fortran routine.

LOGICAL_ARRAY_ARG: Pass a LOGICAL array argument to a Fortran routine.

REAL_ARG: Pass a REAL argument to a Fortran routine.

REAL_ARRAY_ARG: Pass a REAL array argument to a Fortran routine.

WORD_ARG: Pass a WORD argument to a Fortran routine.

WORD_ARRAY_ARG: Pass a WORD array argument to a Fortran routine.

21.7.3 PRIMDAT — Primitive numerical data processing

[SUN/39]

There are a great many routines in this library so only generic names are given and the various possibilities are listed in tables. Variables are specified in routine names by lower-case letters, as shown in Table 1.2.

Value functions — VAL_fs: Apply function f to one or two numbers of type s.
    VAL_sTOt: Convert a number from type s to type t.
Vectorised routines — VEC_fs: Apply function f to one or two vectors of type s.
    VEC_sTOt: Convert a vector from type s to type t.
Numerical functions — NUM_fs: Apply function f to one or two numbers of type s.
    NUM_rs: Apply relation r to two numbers of type s.
    NUM_sTOt: Convert a number from type s to type t.

Symbol	**Meaning**	Table
f	Functions	1.5, 1.6
s, t	Data types	1.3
r	Relations	1.4

Table 1.2: Meaning of symbols used in tables and routine names.

Type Code s or t	**HDS Type**		**Fortran Type**
UB	_UBYTE	(unsigned byte)	BYTE
B	_BYTE	(byte)	BYTE
UW	_UWORD	(unsigned word)	INTEGER∗2
W	_WORD	(word)	INTEGER∗2
I	_INTEGER	(integer)	INTEGER
R	_REAL	(real)	REAL
D	_DOUBLE	(double precision)	DOUBLE PRECISION

Table 1.3: The data type codes used in constants and routine names, and the HDS and VAX Fortran types to which they correspond.

Relational Operation Code r	**Inter-comparison**
EQ	ARG1 .EQ. ARG2
NE	ARG1 .NE. ARG2
GT	ARG1 .GT. ARG2
GE	ARG1 .GE. ARG2
LT	ARG1 .LT. ARG2
LE	ARG1 .LE. ARG2

Table 1.4: The relational operation codes used in the names of NUM functions and the inter-comparisons to which they correspond.

Function Code f	Number of Arguments	Operation performed
ADD	2	addition: ARG1 $+$ ARG2
SUB	2	subtraction: ARG1 $-$ ARG2
MUL	2	multiplication: ARG1 $*$ ARG2
DIV	2	*(floating) division: ARG1 / ARG2
IDV	2	**(integer) division: ARG1 / ARG2
PWR	2	raise to power: ARG1 $**$ ARG2
NEG	1	negate (change sign): $-$ARG
SQRT	1	square root: $\sqrt{\mathrm{ARG}}$
LOG	1	natural logarithm: $\ln(\mathrm{ARG})$
LG10	1	common logarithm: $\log_{10}(\mathrm{ARG})$
EXP	1	exponential: $\exp(\mathrm{ARG})$
ABS	1	absolute (positive) value: $\lvert\mathrm{ARG}\rvert$
NINT	1	nearest integer value to ARG
INT	1	Fortran AINT (truncation to integer) function
MAX	2	maximum: $\max(\mathrm{ARG1}, \mathrm{ARG2})$
MIN	2	minimum: $\min(\mathrm{ARG1}, \mathrm{ARG2})$
DIM	2	Fortran DIM (positive difference) function
MOD	2	Fortran MOD (remainder) function
SIGN	2	Fortran SIGN (transfer of sign) function
	Notes:	*Equivalent to NINT(REAL(ARG1)/REAL(ARG2)) for non-floating quantities
		**Equivalent to AINT(ARG1/ARG2) for floating quantities

Table 1.5: Function codes used in routine names and the operations to which they correspond. The functions shown here are implemented for **all** the numerical data types (type codes UB, B, UW, W, I, R & D).

Function Code f	**Number of Arguments**	**Operation performed**
SIN	1	*sine function: sin(ARG)
SIND	1	**sine function: sin(ARG)
COS	1	*cosine function: cos(ARG)
COSD	1	**cosine function: cos(ARG)
TAN	1	*tangent function: tan(ARG)
TAND	1	**tangent function: tan(ARG)
ASIN	1	*inverse sine function: $\sin^{-1}$(ARG)
ASND	1	**inverse sine function: $\sin^{-1}$(ARG)
ACOS	1	*inverse cosine function: $\cos^{-1}$(ARG)
ACSD	1	**inverse cosine function: $\cos^{-1}$(ARG)
ATAN	1	*inverse tangent function: $\tan^{-1}$(ARG)
ATND	1	**inverse tangent function: $\tan^{-1}$(ARG)
ATN2	2	*Fortran ATAN2 (inverse tangent) function
AT2D	2	**VAX Fortran ATAN2D (inverse tangent) function
SINH	1	hyperbolic sine function: sinh(ARG)
COSH	1	hyperbolic cosine function: cosh(ARG)
TANH	1	hyperbolic tangent function: tanh(ARG)
Notes:	*Argument(s)/result in radians	
	**Argument(s)/result in degrees	

Table 1.6: Function codes used in routine names and the operations to which they correspond. The functions shown here are only implemented for the floating point data types (type codes R & D).

21.7.4 PSX — Posix interface [SUN/121]

Process environment:

PSX_CUSERID: Get the username.

PSX_GETEGID: Get the effective group ID.

PSX_GETENV: Translate an environment variable.

PSX_GETEUID: Get the effective user ID.

PSX_GETGID: Get the real group ID.

PSX_GETPID: Get the process ID.

PSX_GETPPID: Get the process ID of the parent process.

PSX_GETUID: Get the real user ID.

PSX_ISATTY: Determine if a file is a terminal.

PSX_TTYNAME: Get the name of a terminal.

PSX_UNAME: Get information about the host computer system.

Language specific services for C (Fortran version):

Pseudo-Random Numbers — PSX_RAND: Generate a random number.

PSX_SRAND: Set the seed for the random number generator.

Memory Management — PSX_CALLOC: Allocate space for several objects of specified type.

PSX_FREE: Free virtual memory.

PSX_MALLOC: Allocate virtual memory.

PSX_REALLOC: Change the size of an allocated region of virtual memory.

Date and Time — PSX_ASCTIME: Convert a time structure to a character string.

PSX_CTIME: Convert the calendar time to a character string

PSX_LOCALTIME: Convert the value returned by PSX_TIME to individual values.

PSX_TIME: Get the current calendar time.

21.7.5 SLALIB — Positional astronomy and time [SUN/67]

Some of the items listed below are implemented as subroutines and some as functions. Also, some items are implemented in both single-precision and double-precision forms. For brevity, only the single-precision forms are listed.

String decoding — SLA_INTIN: Convert free-format string into integer.

SLA_FLOTIN: Convert free-format string into floating-point number.

SLA_AFIN: Convert free-format string from deg,arcmin,arcsec to radians.

Sexagesimal conversion — SLA_CTF2D: Hours, minutes, seconds to days.

SLA_CD2TF: Days to hours, minutes, seconds.

SLA_CTF2R: Hours, minutes, seconds to radians.

SLA_CR2TF: Radians to hours, minutes, seconds.

SLA_CAF2R: Degrees, arcminutes, arcseconds to radians.

SLA_CR2AF: Radians to degrees, arcminutes, arcseconds.

Angles, Vectors and Rotation matrices — SLA_RANGE: Normalise angle into range $\pm\pi$.

SLA_RANORM: Normalise angle into range $0-2\pi$.

SLA_CS2C: Spherical coordinates to $[x, y, z]$.

SLA_CC2S: $[x, y, z]$ to spherical coordinates.

SLA_VDV: Scalar product of two 3-vectors.

SLA_VXV: Vector product of two 3-vectors.

SLA_VN: Normalise a 3-vector also giving the modulus.

SLA_SEP: Angle between two points on a sphere.

SLA_BEAR: Direction of one point on a sphere seen from another.

SLA_EULER: Form rotation matrix from three Euler angles.

SLA_AV2M: Form rotation matrix from axial vector.

SLA_M2AV: Determine axial vector from rotation matrix.

SLA_MXV: Rotate vector forwards.

SLA_IMXV: Rotate vector backwards.

SLA_MXM: Product of two 3x3 matrices.

SLA_CS2C6: Convert position and velocity in spherical coordinates to Cartesian coordinates.

SLA_CC62S: Convert position and velocity in Cartesian coordinates to spherical coordinates.

Calendars — SLA_CLDJ: Gregorian Calendar to Modified Julian Date.

SLA_CALDJ: Gregorian Calendar to Modified Julian Date, permitting century default.

SLA_DJCAL: Modified Julian Date to Gregorian Calendar, convenient for formatted output.

SLA_DJCL: Modified Julian Date to Gregorian Year, Month, Day, Fraction.

SLA_CALYD: Calendar to year and day in year.

SLA_EPB: Modified Julian Date to Besselian Epoch.

SLA_EPB2D: Besselian Epoch to Modified Julian Date.

SLA_EPJ: Modified Julian Date to Julian Epoch.

SLA_EPJ2D: Julian Epoch to Modified Julian Date.

Timescales — SLA_GMST: Conversion from Universal Time to sidereal time.

SLA_EQEQX: Equation of the equinoxes.

SLA_DAT: Offset of Atomic Time from UT: TAI−UTC.

SLA_DTT: Offset of "Ephemeris Time" from UT: TDT−UTC.

SLA_RCC: Relativistic clock correction: TDB−TDT.

Precession and Nutation — SLA_NUT: Nutation matrix.

SLA_NUTC: Longitude and obliquity components of nutation, and mean obliquity.

SLA_PREC: Precession matrix.

SLA_PRENUT: Combined precession/nutation matrix.

SLA_PREBN: Precession matrix, old system.

SLA_PRECES: Precession, in either the old or the new system.

Proper motion — SLA_PM: Adjust for proper motion.

FK4/5 conversions — SLA_FK425: Convert B1950.0 FK4 star data to J2000.0 FK5.

SLA_FK45Z: Convert B1950.0 FK4 position to J2000.0 FK5 assuming zero proper motion in an inertial frame and no parallax.

SLA_FK524: Convert J2000.0 FK5 star data to B1950.0 FK4.

SLA_FK54Z: Convert J2000.0 FK5 position to B1950.0 FK4 assuming zero proper motion and no parallax.

SLA_DBJIN: Like SLA_DFLTIN but with extensions to accept leading 'B' and 'J'.

SLA_KBJ: Select epoch prefix 'B' or 'J'.

SLA_EPCO: Convert an epoch into the appropriate form – 'B' or 'J'.

Elliptic aberration — SLA_ETRMS: E-terms.

SLA_SUBET: Remove the E-terms.

SLA_ADDET: Add the E-terms.

Geocentric coordinates — SLA_OBS: Interrogate list of observatory parameters.

SLA_GEOC: Convert geodetic position to geocentric.

SLA_PVOBS: Position and velocity of observatory.

Apparent and observed place — SLA_MAP: Mean place to geocentric apparent place.

SLA_MAPPA: Precompute mean to apparent parameters.

SLA_MAPQK: Mean to apparent using precomputed parameters.

SLA_MAPQKZ: Mean to apparent using precomputed parameters, for zero proper motion, parallax and radial velocity.

SLA_AMP: Geocentric apparent place to mean place.

SLA_AMPQK: Apparent to mean using precomputed parameters.

SLA_AOP: Apparent place to observed place.

SLA_AOPPA: Precompute apparent to observed parameters.

SLA_AOPPAT: Update sidereal time in apparent to observed parameters.

SLA_AOPQK: Apparent to observed using precomputed parameters.

SLA_OAP: Observed to apparent.

SLA_OAPQK: Observed to apparent using precomputed parameters.

SLA_ZD: $[h, \delta]$ to zenith distance.

SLA_PA: $[h, \delta]$ to parallactic angle.

Refraction and air mass — SLA_REFRO: Change in zenith distance due to refraction.

SLA_REFCO: Constants for simple refraction model.

SLA_REFZ: Unrefracted to refracted ZD, simple model.

SLA_REFV: Unrefracted to refracted $[Az, El]$ vector, simple model.

SLA_AIRMAS: Air mass.

Ecliptic coordinates — SLA_ECMAT: Equatorial to ecliptic rotation matrix.

SLA_EQECL: J2000.0 'FK5' to ecliptic coordinates.

SLA_ECLEQ: Ecliptic coordinates to J2000.0 'FK5'.

Galactic coordinates — SLA_EG50: B1950.0 'FK4' to galactic.

SLA_GE50: Galactic to B1950.0 'FK4'.

SLA_EQGAL: J2000.0 'FK5' to galactic.

SLA_GALEQ: Galactic to J2000.0 'FK5'.

Supergalactic coordinates — SLA_GALSUP: Galactic to supergalactic.

SLA_SUPGAL: Supergalactic to galactic.

Ephemerides — SLA_EARTH: Approximate heliocentric position and velocity of the Earth.

SLA_EVP: Barycentric and heliocentric velocity and position of the Earth.

SLA_MOON: Approximate geocentric position and velocity of the Moon.

SLA_RVEROT: Velocity component due to rotation of the Earth.

SLA_ECOR: Components of velocity and light time due to Earth orbital motion.

SLA_RVLSR: Velocity component due to solar motion wrt LSR.

SLA_RVGALC: Velocity component due to rotation of the Galaxy.

SLA_RVLG: Velocity component due to rotation and translation of the Galaxy, relative to the mean motion of the local group.

Astrometry — SLA_S2TP: Transform spherical coordinates into tangent plane.

SLA_TP2S: Transform tangent plane coordinates into spherical.

SLA_PCD: Apply pincushion/barrel distortion.

SLA_UNPCD: Remove pincushion/barrel distortion.

SLA_FITXY: Fit a linear model to relate two sets of $[x, y]$ coordinates.

SLA_PXY: Compute predicted coordinates and residuals.

SLA_INVF: Invert a linear model.

SLA_XY2XY: Transform one $[x, y]$.

SLA_DCMPF: Decompose a linear fit into scales *etc.*

Numerical methods — SLA_SMAT: Matrix inversion and solution of simultaneous equations.

SLA_SVD: Singular value decomposition of a matrix.

SLA_SVDSOL: Solution from given vector plus SVD.

SLA_SVDCOV: Covariance matrix from SVD.

SLA_RANDOM: Generate pseudo-random real number in the range $0 \leq x < 1$.

SLA_GRESID: Generate pseudo-random normal deviate ($\equiv$ 'Gaussian residual').

Real-time — SLA_WAIT: Interval wait.

21.7.6 TRANSFORM — Coordinate transformation [SUN/61]

TRN_ANNUL: Annul compiled mapping.

TRN_APND: Append transformation.

TRN_CLOSE: Close the TRANSFORM facility.

TRN_COMP: Compile transformation.

TRN_GTCL: Get classification.

TRN_GTCLC: Get compiled classification.

TRN_GTNV: Get numbers of variables.

TRN_GTNVC: Get numbers of compiled variables.

TRN_INV: Invert transformation.

TRN_JOIN: Concatenate transformations.

TRN_NEW: Create new transformation.

TRN_PRFX: Prefix transformation.

TRN_PTCL: Put classification.

TRN_STOKx: Substitute token.

TRN_TR1x: Transform 1-d data.

TRN_TR2x: Transform 2-d data.

TRN_TRNx: Transform general data.