SG/5.2

Starlink Project
Starlink Guide 5.2

J A Bailey[1]
A J Chipperfield

9th June 1998

ICL

The Interactive Command Language for ADAM Version 1.5-6 User's Guide

[1]Anglo-Australian Observatory

Abstract

ICL is a language designed to provide a programmable user interface to an astronomical data reduction or data acquisition system. It is the primary user interface for the ADAM software environment.

This document is a re-formatted version of SG/5.1 – the text has not changed. For information on ICL for Unix, ICL help is more reliable.

Contents

I Introduction to ICL **1**

1 Introduction **2**
1.1 What is ICL? . . . 2
1.2 ICL Documentation . . . 2
1.3 ICL and FORTRAN . . . 2
1.4 History and Support . . . 2

2 Getting Started **4**
2.1 Direct Mode . . . 4
2.2 Entering Commands . . . 4
2.3 The Immediate Statement . . . 4
2.4 The Assignment Statement . . . 5
2.5 Expressions . . . 5
2.5.1 Values . . . 5
2.5.2 Constants . . . 5
2.5.3 Variables . . . 6
2.5.4 Operators . . . 6
2.5.5 Functions . . . 7
2.6 Formatting Numbers For Output . . . 7
2.7 Commands . . . 7
2.7.1 Command Format . . . 8
2.7.2 Summary of Rules for Command Parameters . . . 8

3 ICL Procedures **10**
3.1 Direct Entry of Procedures . . . 10
3.2 Running a Procedure . . . 10
3.3 Listing Procedures . . . 11
3.4 Editing Procedures . . . 11
3.5 Direct Commands During Procedure Entry . . . 12
3.6 Saving and Loading Procedures . . . 12
3.7 Control Structures . . . 13
3.7.1 The IF structure . . . 13
3.7.2 The LOOP Structure . . . 14
3.8 Prompting for Procedure Parameters . . . 15
3.9 Variables in Procedures . . . 16
3.10 Tracing Procedure Execution . . . 16
3.11 Running ICL as a Batch Job . . . 17

4 Input/Output **19**
4.1 Terminal I/O . . . 19
4.2 Text File I/O . . . 19
4.3 Screen Mode . . . 20
4.4 Keyboard Facilities in Screen Mode . . . 21

5 Access to DCL **23**
5.1 The $ Command . . . 23
5.2 The SPAWN Command . . . 23
5.3 Changing the Default Directory . . . 24
5.4 Allocating and Mounting Tape Drives . . . 24

6 Errors and Exceptions **26**
6.1 ICL Exceptions . . . 26
6.2 Exception Handlers . . . 27
6.3 Keyboard Aborts . . . 29
6.4 SIGNAL command . . . 29

7 Extending ICL **30**
7.1 Login files . . . 30
7.2 The DEFSTRING Command . . . 31
7.3 Hidden Procedures . . . 31
7.4 The DEFPROC Command . . . 31
7.5 Defining Additional Help Topics . . . 31
7.6 Other Ways of Defining Commands . . . 32

II ICL and ADAM **33**

8 Introduction to ADAM **34**
8.1 What is ADAM? . . . 34
8.2 The Role of the Command Language . . . 34

9 Using ADAM for Data Reduction **36**
9.1 A-tasks . . . 36
9.2 Monoliths . . . 36
9.3 Running KAPPA . . . 36
9.4 ADAM Parameter Format . . . 38
9.5 KAPPA from Procedures . . . 39
9.6 Error Reports . . . 39

10 Writing ADAM tasks **40**
10.1 Introduction . . . 40
10.2 Compiling and Linking . . . 40
10.3 Tasks with Parameters . . . 41
10.4 STATUS and error handling . . . 42
10.5 Returning values to ICL . . . 43
10.6 Graphics with ADAM . . . 44
10.7 Accessing Data . . . 45

11 ADAM as a Data Acquisition Environment **48**
11.1 Instrumentation Tasks . . . 48
11.2 D-tasks and C-tasks . . . 48
11.3 Task Loading . . . 48
11.4 Killing Tasks . . . 49
11.5 The ADAM message system . . . 49
11.6 An Example . . . 49
11.6.1 The I-task Interface File . . . 49
11.6.2 Running PHOTOM . . . 50
11.6.3 Supplying Parameters in the Obey Message . . . 50
11.6.4 Cancelling Actions . . . 50

11.6.5 Missing Parameters . . . 50
11.6.6 The GET command . . . 51
11.6.7 The OBEYW command . . . 51
11.6.8 Multiple Concurrent Actions . . . 51

A ICL Functions **53**

B ICL Commands **58**
B.1 ALLOC . . . 58
B.2 ALOAD . . . 58
B.3 APPEND . . . 58
B.4 CHECKTASK . . . 59
B.5 CLEAR . . . 59
B.6 CLOSE . . . 59
B.7 CREATE . . . 59
B.8 CREATEGLOBAL . . . 59
B.9 $ (DCL) . . . 60
B.10 DEALLOC . . . 60
B.11 DEFAULT . . . 60
B.12 DEFHELP . . . 60
B.13 DEFINE . . . 61
B.14 DEFPROC . . . 61
B.15 DEFSHARE . . . 62
B.16 DEFSTRING . . . 62
B.17 DEFTASK . . . 62
B.18 DEFUSER . . . 63
B.19 DELETE . . . 63
B.20 DISMOUNT . . . 63
B.21 DUMPTASK . . . 64
B.22 EDIT . . . 64
B.23 ENDOBEY . . . 64
B.24 EXIT . . . 64
B.25 GET . . . 64
B.26 GETGLOBAL . . . 65
B.27 GETNBS . . . 65
B.28 GETPAR . . . 65
B.29 HELP . . . 65
B.30 INPUT . . . 66
B.31 INPUTI . . . 66
B.32 INPUTL . . . 66
B.33 INPUTR . . . 66
B.34 KEY . . . 67
B.35 KEYTRAP . . . 67
B.36 KEYUSER . . . 67
B.37 KILL . . . 67
B.38 KILLDCL . . . 68
B.39 KILLW . . . 68
B.40 LIST . . . 68
B.41 LOAD . . . 68
B.42 LOADD . . . 68
B.43 LOADW . . . 69
B.44 MOUNT . . . 69
B.45 NOREP . . . 69
B.46 OBEYW . . . 69
B.47 OPEN . . . 70

B.48 PRINT . . . 70
B.49 PROCS . . . 70
B.50 PUTNBS . . . 70
B.51 READ . . . 70
B.52 READI . . . 71
B.53 READL . . . 71
B.54 READR . . . 71
B.55 REPFILE . . . 71
B.56 REPORT . . . 72
B.57 SAVE . . . 72
B.58 SAVEINPUT . . . 72
B.59 SEND . . . 72
B.60 SET . . . 73
B.60.1 SET ATTRIBUTES . . . 73
B.60.2 SET AUTOLOAD, SET NOAUTOLOAD . . . 73
B.60.3 SET CHECKPARS, SET NOCHECKPARS . . . 73
B.60.4 SET EDITOR . . . 73
B.60.5 SET HELPFILE . . . 74
B.60.6 SET MESSAGES, SET NOMESSAGES . . . 74
B.60.7 SET PRECISION . . . 74
B.60.8 SET PROMPT . . . 74
B.60.9 SET SAVE, SET NOSAVE . . . 74
B.60.10 SET SCREEN, SET NOSCREEN . . . 74
B.60.11 SET TRACE, SET NOTRACE . . . 74
B.61 SETGLOBAL . . . 75
B.62 SETPAR . . . 75
B.63 SIGNAL . . . 75
B.64 SPAWN . . . 76
B.65 STARTOBEY . . . 76
B.66 TASKS . . . 76
B.67 VARS . . . 76
B.68 WAIT . . . 76
B.69 WRITE . . . 77

C ICL Exceptions **78**

D ICL Syntax **81**

Part I

Introduction to ICL

Chapter 1
Introduction

1.1 What is ICL?

The Interactive Command Language (ICL) is a language designed to provide a programmable user interface to an astronomical data reduction or data acquisition system. It is the primary user interface for the ADAM software environment and its use with ADAM is described in Part II.

ICL is in some ways similar to a high level programming language such as Fortran or Pascal, but it has some important differences.

- It is a *command* language. One of its main uses is to enable the typing of commands with few restrictions on the possible command format For example ICL can be used to run the FIGARO data reduction system and it is possible to type FIGARO commands in exactly the same format as was previously used from DCL.
- It is an *interactive* language. ICL provides a complete environment for entering, editing and debugging programs, rather than relying on external editors, linkers *etc.*
- ICL can be used as a programming language, but it is intended for writing relatively simple and straightforward programs. It's requirements are different from those of most modern programming languages, which are designed for the needs of big software projects such as writing operating systems or controlling missiles. ICL is designed to make simple programs easy to write.

1.2 ICL Documentation

This users' guide provides an introduction to ICL for beginners, and should provide the information needed to get started with it. The various appendices give further details on many aspects of the language.

ICL also provides an on-line help system. Simply type HELP and a list of topics will be displayed (it works in exactly the same way as the DCL help system).

1.3 ICL and FORTRAN

The reader of this manual is assumed to be familiar with programming in FORTRAN, and the manual will compare ICL features with the corresponding features of FORTRAN where appropriate.

1.4 History and Support

ICL was written by Jeremy Bailey, working at the Anglo-Australian Observatory and subsequently at the Joint Astronomy Centre, Hilo. Responsibility for support has now passed to the ADAM Support Group,

part of the Science and Engineering Research Council's Starlink Project at the Rutherford Appleton Laboratory.

Chapter 2
Getting Started

2.1 Direct Mode

In this section we describe how to run ICL in direct mode and use its variables, expressions *etc.* which enable us to use the VAX as a very expensive electronic calculator. In direct mode we type in commands and they are executed immediately. The alternative to direct mode is the use of procedures, in which the commands are entered into a procedure, and subsequently executed by running the procedure.

ICL is started up by using the command ICL. After a short delay ICL will respond by displaying any startup messages which have been set, followed by the prompt `ICL>`, meaning it is now ready to accept a command.

```
$ ICL

  Interactive Command Language   -   Version 1.5-6

  - Type HELP [command] for help on ICL and its commands

ICL>
```

2.2 Entering Commands

When entering commands in ICL all the normal command line editing facilities available in DCL may be used. The Up arrow or CTRL/B key 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.

2.3 The Immediate Statement

The first ICL statement we will introduce is the immediate statement. This is used to make ICL do simple calculations. It simply consists of an equals sign (=) followed by an expression, and causes the value of the expression to be printed on the terminal.

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

ICL arithmetic expressions are very similar to expressions in FORTRAN, and will be discussed in more detail later.

2.4 The Assignment Statement

The other commonly used statement is the assignment statement, which is exactly the same as the FORTRAN assignment statement, and is used to assign a value to a variable.

```
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.

2.5 Expressions

Expressions are built up by operating on *values* using *operators*. The values can be represented by either *constants* or *variables* or can be the result of a *function* operating on another expression.

2.5.1 Values

ICL operates on values of four different types:

- Integer
- Real
- Logical
- String

The integer, real and logical types are the same as their FORTRAN equivalents (the ICL real type is strictly equivalent to FORTRAN's double precision being stored with about 16 decimal digits of precision. The String type represents strings of characters and is similar to the FORTRAN CHARACTER*n type.

2.5.2 Constants

Values of all four types can be represented by appropriate constants. Integer and real 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 preceding the value with `%B`, `%O` or `%X`.

Logical constants are typed 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.

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).

2.5.3 Variables

Variables in ICL 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 (*i.e.* two variable names which are the same in the first 15 characters, but differ in subsequent characters refer to the same variable).

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, which is either derived implicitly from the first letter of the name, or is explicitly specified in a declaration.

In ICL names do not have types, only values have types. A variables 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 sequence 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 we do not have to declare the variables we use which helps to keep programs simple.[1]

2.5.4 Operators

The operators which are used to build up expressions are listed in the following table.

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. In general however, it is good practice to use parentheses to clarify any situation in which the order of evaluation is in doubt.

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 numeric value 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).

The operator & performs string concatenation – operands with numeric values will be converted to strings. The : operator is used for formatting numbers into character strings as described below.

[1] The disadvantage is that ICL cannot usually spot cases where we accidentally mistype the name of a variable, 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.

[2] But different from those in Pascal

2.5.5 Functions

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

2.6 Formatting Numbers For 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. Thus 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, and four 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.

```
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
```

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. 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 various bases.

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

2.7 Commands

We have now met two of the three statement types available in direct mode, the immediate and assignment statements. We now introduce the third and most important one, the command. Commands are used for three purposes.

- To provide features of the ICL system itself — These are commands such as PRINT, EDIT, LIST *etc.*
- To call procedures written in ICL.
- To provide the commands of the system ICL is being used to run — In this case ADAM.

2.7.1 Command Format

An ICL command consists of a command name, which is formed using exactly the same rules as we described earlier for variable names, followed by optionally, one or more parameters. Parameters can take three forms:

- An expression or variable name enclosed in parentheses.
- A string enclosed in quotes.
- Any sequence of characters not including a space, quote, comma or left parenthesis.

The parameters may be separated by commas, or by one or more spaces.

The first form of the parameter is used when we want to pass the value of an expression to the command, or we want to give the command a variable into which it will return a value. The other two forms both pass a string.

We can illustrate these various cases by using the PRINT command, which prints its parameters on the terminal.

```
ICL> X=1.234
ICL> PRINT (X)
1.23400
ICL> PRINT 'HELLO'
HELLO
ICL> PRINT HELLO
HELLO
```

In most cases therefore we do not need to use the quoted form of string parameters because the simpler form will work. We need the quoted form of strings for those cases in which we need to include a left parenthesis, or spaces in the string. Here is an example with several parameters.

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

What is happening here is that, since spaces are parameter separators, `'The'`, `'Square'`, `'Root'` and `'of'` are all received by PRINT as independent parameters. However PRINT simply concatenates all its parameters, with a space between each pair, and thus the result is the string just as we typed it. Many other ICL commands which accept strings work in this way. This means that strings with *single* spaces do not usually need quotes when used as command parameters.

2.7.2 Summary of Rules for Command Parameters

The rules for specifying command parameters can be summarized as follows.

- To pass the value of an expression (which might be just a single number), or the name of a variable, the expression or variable should be placed 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.

FORTRAN programmers may find it useful to note that the parenthesized form of the command parameters is exactly equivalent to the normal FORTRAN method of passing parameters to a subroutine. Thus the ICL command

```
COMMAND (A) (B) (C)
```

is equivalent to the FORTRAN

```
CALL COMMAND (A, B, C)
```

Any parameter can always be passed in this way. The other forms of parameter simply provide a convenient way of handling the common case of passing a string constant.

These rules may appear somewhat confusing specified in this way, but what they achieve is to allow us to type many familiar commands in the way we are used to. Thus the following are valid ICL commands:

```
ICL> $DIR/SIZE/DATE/SINCE=TODAY

ICL> SPLOT MYSPECT LOW=10 HIGH=100 \
```

Chapter 3
ICL Procedures

An ICL Procedure is essentially the equivalent of a Subroutine in FORTRAN. It allows us to write a sequence of ICL statements which can be run with a single command. The procedure may have a number of parameters which are used to pass values to the procedure, and return values from it.

3.1 Direct Entry of Procedures

To enter an ICL procedure we type a PROC command which specifies the name of the procedure, and the names of any of its parameters. ICL then returns a new prompt using the name of the procedure rather than `ICL>` to show that we are in procedure entry 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.

```
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>
```

The line beginning with { is a comment which will be ignored by ICL when executing the procedure. The closing } is not necessary but looks neater.

3.2 Running a Procedure

To run the procedure we have entered we use the command format described earlier, using the procedure name as the command, and adding any parameters required.

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

But we can also specify the parameter without parentheses:

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

What has happened here is that instead of the numeric value 2, we have passed the string '2' as the parameter. However, the SQRT function requires a numeric argument, so converts this string to a number. Thus, the free approach to type conversion, means that in many cases the rules for command parameters described in the Section 2.7.2 can be relaxed.

```
ICL> Y=3
ICL> SQUARE_ROOT Y
The Square Root of Y is  1.732051
```

In this case the parameter passed was the string 'Y', but once again this was converted to a numeric value for the SQRT function.

3.3 Listing Procedures

The LIST command can be used to list 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>
```

To find the names of all the current procedures use the command PROCS.

```
ICL> PROCS

SQUARE_ROOT

ICL>
```

3.4 Editing Procedures

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

```
ICL> EDIT SQUARE_ROOT
```

would 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. All these editors are described in DEC's documentation.

When editing a procedure there are two possible options.

- We can leave the name of the procedure unchanged, but edit the code. In this case we create a new version of the procedure, which replaces the old one when we exit from the editing session.
- We can change the name of the procedure by editing the PROC statement at the start of the procedure. This creates a new procedure with the new name, and leaves the old procedure unchanged.

It is possible to enter procedures completely using the editors. However it is recommended that procedures be entered originally using direct entry. The advantage is that during direct entry, any errors will be detected immediately. Thus if we mistype the PRINT line in the above example we get the following error message:

```
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 found the `'is'` string when a right parenthesis was expected.

Following such an error message we 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.

3.5 Direct Commands During Procedure Entry

It is sometimes useful to have a command directly executed 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 us on-line help information we might need to complete the procedure. If the % was omitted the HELP command would be included as part of the procedure.

3.6 Saving and Loading Procedures

Procedures created in the above way will only exist for the duration of an ICL session. If we 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 our SQUARE_ROOT procedure on disk we would use:

```
ICL> SAVE SQUARE_ROOT
```

To load it again, probably in a subsequent ICL session we 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. With LOAD however 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 all the current procedures in a single file with the name SAVE.ICL. These procedures may then be reloaded by the command:

```
ICL> LOAD SAVE
```

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

3.7 Control Structures

The ICL control structures provide a means of controlling the flow of execution within a procedure. Unlike the statements we have met so far these can only be used within a procedure and are not accepted in direct mode. There are two types of control structure:

- The IF or conditional structure.
- The LOOP structure.

3.7.1 The IF structure

The IF structure 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, and the following statements are then executed. If none of the expressions are true the statements following ELSE are executed.

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

```
IF expression
    statements
ENDIF
```

The following example illustrates the use of the IF structure, and shows how one IF structure may be nested within another.

```
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
```

3.7.2 The LOOP Structure

The LOOP structure is used to repeatedly execute a group of statements. It has three different forms, the simplest being as follows:

```
LOOP
    statements
END LOOP
```

This form sets up an infinite loop, but an additional statement, BREAK, may be used to terminate the loop. BREAK would, of course, normally have to be inside an IF structure.

```
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
```

As it is frequently required to loop over a sequential range of numbers in this way, 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 integer values. Using this form of the LOOP statement we 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.

The third form of the LOOP structure allows the setting up of loops which terminate on any general 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 we 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 the user for an answer to a question (*e.g.* yes or no) 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
```

3.8 Prompting for Procedure Parameters

Normally a procedure will not be obeyed unless the required number of parameters is provided – the TOOFEWPARS exception is signalled. However, this check can be switched off using the NOCHECKPARS command. Then the procedure will be executed with the parameter being undefined. This fact can be used to write a procedure which will prompt for an undefined parameter, *e.g.*:

```
PROC SQUARE_ROOT X
{ An ICL procedure to print the square root of a number,   }
{ prompting if the number is not given on the command line }
  IF UNDEFINED(X)
    INPUT 'Give the value of X: ' (X)
  ENDIF
  PRINT The Square Root of (X) is (SQRT(X))
END PROC
```

Note that only trailing parameters may be omitted

3.9 Variables in Procedures

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
1.2345
ICL> =X
       1
ICL>
```

When we run the procedure FRED we get the value of the variable X in 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 we can use procedures freely without having to worry about any possible side effects of the procedure on variables outside it.

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.

The command VARS is used to list all the variables of a procedure. It has one parameter, which is the name of the procedure. If the parameter is omitted, then the outer level variables, *i.e.* those that are not part of any procedure are listed. Thus in the previous example:

```
ICL> VARS FRED
                X  REAL     1.23450E+00
ICL> VARS
                X  INTEGER          1
ICL>
```

VARIABLE is a function whose result is the value of a given variable in a given procedure:

```
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 a procedure finishes execution, and if the procedure is executed a second time, they will retain their values from the first time through the procedure.

3.10 Tracing Procedure Execution

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 either be issued 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.

3.11 Running ICL as a Batch Job

It is sometimes useful to run one or more ICL procedures as a batch job. It is quite easy to set this up 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 as mentioned earlier, a LOAD file may include direct commands as well as procedures. In order to create a Batch job we must set up a file which contains all the procedures we want, 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 our 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 adding the additional direct commands. Supposing this file is called TABLE.ICL. To create a batch job we also need a command file, which we could call TABLE.COM which 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 the top level directory and will contain the output from the batch job. A /OUTPUT qualifier can be used to specify a different file name or directory for it.

Chapter 4
Input/Output

4.1 Terminal I/O

We have already met the PRINT command for output to the terminal. The analogous commands for terminal input are called INPUT, INPUTR, INPUTI and INPUTL. INPUT reads a line of text from the terminal into a string variable and has the form:

```
ICL> INPUT Enter your name>  (name)
```

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

INPUTR, INPUTI and INPUTL are used to input real, integer or logical values. A single line of text may be used to supply values for more than one variables so these commands have the form:

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

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.

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

4.2 Text File I/O

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

```
CREATE MYFILE
```

will create a file called MYFILE and open it for output. MYFILE is the name used within ICL for the file. The file will appear in your default VMS directory as MYFILE.DAT.

The VMS file name may be specified explicitly by adding a second parameter to the CREATE, OPEN or APPEND command.

```
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 the file is appended to the existing contents.

A line of text is written to a text 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 uses these procedures to read an input 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

{  Loop copying 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]

4.3 Screen Mode

ICL's 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 screen 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
```

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

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 available in this way.

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

```
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:

```
CLEAR 6 10
```

clears lines 6 to 10 of the screen.

4.4 Keyboard Facilities in Screen Mode

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

```
KEY  PF1  PROCS#
```

defines the PF1 key so as to issue 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 of 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.

KEYTRAP and INKEY 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. 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. The KEYVAL function may be used to obtain the value from the key name.

```
{  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
```

The command

```
KEYOFF keyname
```

may be used to turn off trapping of a key.

Chapter 5
Access to DCL

When working from 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.

5.1 The $ Command

The command $ allows any DCL command to be issued from inside ICL. It's form is simply:

```
$ dcl_command
```

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

```
$ COPY *.DST DATADIR:*.DST
$ RUN MYPROGRAM
```

There is one restriction — we must use a complete DCL command. We couldn't for example, just type $ COPY and let DCL prompt us for the two file specifications as we could from the DCL $ prompt. Apart from this any command acceptable to DCL can be issued in this way.

'DCL' may be used as an alternative to the $ command. Thus the above example could also have been written as:

```
DCL COPY *.DST DATADIR:*.DST
DCL RUN MYPROGRAM
```

5.2 The SPAWN Command

There is also a way round the restriction mentioned above. This is to use the command SPAWN rather than the $ command. For example:

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

in which case we get the From: and To: prompts just as we do in normal DCL. The disadvantage of SPAWN is that it is normally much slower. This is because SPAWN creates a new subprocess to issue each command, whereas DCL creates a permanent subprocess in which all commands are issued.

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

5.3 Changing the Default Directory

It might seem that the above facilities provide all we need. Unfortunately things are not that simple. The problem is that VMS only provides the facility to issue a DCL command in a subprocess, not in the process we are actually running ICL in. Thus although we can issue any DCL command we cannot issue DCL commands in the process we are running ICL in. In many cases this does not matter, the command will have the same effect whatever process it is issued from.

However, this is not always the case. One example is changing the default directory — this can be done using the ICL 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` (which may be abbreviated to `DEF`) 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 DCL SET DEFAULT command.

5.4 Allocating and Mounting Tape Drives

Similar problems occur when allocating and mounting tape drives. $ 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 the tape is to be processed using a FIGARO command it must be allocated to the process running ICL.

A set of commands has been provided for this purpose as follows:

command	abbreviation	function
ALLOC dev	ALL	allocate a device
MOUNT dev	MOU	mount a device
DISMOUNT dev	DISMOU	dismount a device
DEALLOC dev	DEALL	deallocate a device

Mount performs a MOUNT/FOREIGN at the tapes initialized density. It does not provide the many qualifiers of the DCL command. There are several additional optional parameters for some of these commands. ALLOC may specify a generic name, and the name of the device actually allocated will be returned in the optional second parameter.

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

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

```
ICL> DISMOU MTA1 NOUNLOAD
```

Chapter 6
Errors and Exceptions

6.1 ICL Exceptions

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

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

We 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 appendix C.

If the error occurs within a procedure the message contains a little more information. As an example, if we use our square root procedure of Section 3.11 with an invalid value we 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. If we run the following procedure

```
PROC TABLE
{  Print a table of Square roots from 5 down to -5  }
   LOOP FOR I = 5 TO -5 STEP -1
     SQUARE_ROOT (I)
   END LOOP
END PROC
```

we get

```
ICL> TABLE
The Square Root of 5 is 2.236068
The Square Root of 4 is 2
The Square Root of 3 is 1.732051
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>
```

6.2 Exception Handlers

It is often useful to be able to modify the default behaviour on an error condition. We 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 5 is 2.236068
The Square Root of 4 is 2
The Square Root of 3 is 1.732051
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
The Square Root of -3 is 1.732051i
The Square Root of -4 is 2i
The Square Root of -5 is 2.236068i
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 are included in a procedure following the normal code of the procedure 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 same procedure in which the exception occurred, but could be in a procedure further up in the chain of calls. In our example we could put an exception handler for SQUROONEG in TABLE rather than in SQUARE_ROOT.

```
PROC TABLE
{  Print a table of Square roots from 5 down to -5  }
   LOOP FOR I = 5 TO -5 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 5 is 2.236068
The Square Root of 4 is 2
The Square Root of 3 is 1.732051
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
```

6.3 Keyboard Aborts

One exception which is commonly encountered is that which results when a Control-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 control-C is typed.

6.4 SIGNAL command

The exceptions described up to now have all been generated internally by the ICL system, or in the case of CTRLC are 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 using the SIGNAL command. This 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 appendix C. 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.

Chapter 7
Extending ICL

If you use ICL frequently you will find it convenient to define your own commands. We have already met one way of defining new commands. This is to write an ICL procedure as described in Chapter 3. There are several other ways of defining new commands which are described in this chapter. If you have many commands which you want to use frequently it is convenient to put these into a login file which will be automatically loaded each time ICL is started up.

7.1 Login files

An ICL login file works in exactly the same way as your DCL LOGIN.COM file. ICL uses the logical name ICL_LOGIN to locate your login file so if you create a file called LOGIN.ICL in your top level directory you can use use the following DCL DEFINE command (which you should put 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 file will then be loaded automatically whenever you start up ICL and can include procedures, definitions of commands, or indeed, any valid ICL commands. Below is an example of a login file which illustrates some of the facilities which may be used.

```
{  ICL  Login File  }

{  Define TYPE command  }

DEFSTRING  T(YPE)  $ TYPE

{  Define EDIT command  }

HIDDEN PROC EDIT name
   IF INDEX(name,'.') = 0
      #EDIT (name)
   ELSE
      DCL EDIT (name)
   ENDIF
END PROC

{  Login Message  }

PRINT
PRINT   Starting ICL at (TIME()) on (DATE())
PRINT
```

7.2 The DEFSTRING Command

The first entry in the file defines the DCL TYPE command so that it is accessible directly from ICL without having to do enter DCL TYPE. This is done using the DEFSTRING command which defines an equivalence string for a command. In this case the command is TYPE and the equivalence string is DCL TYPE. The notation T(YPE) specifies possible abbreviations for the TYPE command, indicating that we could actually use T, TY or TYP instead of TYPE in full.

7.3 Hidden Procedures

The definition of the EDIT command is done in a different way. Since EDIT is used within ICL to edit procedures, if we just used DEFSTRING to define EDIT as $ EDIT we would lose the ability to edit ICL procedures. The EDIT command would always edit VMS files.

The procedure used to redefine EDIT gets round this by testing for the existence of a dot in the name 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 is assumed 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. 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 accidentally deleted.

7.4 The DEFPROC Command

If you have many procedures of this type you may not wish to include them in full in your login file, because this will require them all to be compiled when ICL starts up and may therefore slow down the start up process. The DEFPROC command allows you to define commands which run ICL procedures, but with the procedures only being compiled when they are required. For example if the EDIT procedure described above was put in the source file EDIT.ICL we could put the following DEFPROC command in the login file.

```
DEFPROC ED(IT) EDIT.ICL
```

This command specifies that the command EDIT (with minimum abbreviation ED) is to run the procedure EDIT in source file EDIT.ICL.

The procedure will not be loaded and compiled until the first time the EDIT command is issued.

7.5 Defining Additional Help Topics

ICL includes a HELP command which provides on line documentation on ICL itself. Using the DEFHELP command it is possible to extend the 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 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.

7.6 Other Ways of Defining Commands

The DEFUSER command allows an ICL command to be defined to call a FORTRAN subroutine. In order to make this work you have to link the subroutine into a shareable image (The VAX/VMS linker manual explains how to do this). The subroutine should have a single character string parameter in which it will receive the command line parameters of the original ICL command (after substitution of any bracketed expressions).

An easier way of achieving a similar result is to write your FORTRAN subroutine as an ADAM A-task, and use the DEFINE command as described in the next section.

Part II

ICL and ADAM

Chapter 8
Introduction to ADAM

8.1 What is ADAM?

ADAM (Astronomical Data Acquisition Monitor) consists of a number of facilities which can be combined in a toolkit approach to support a range of software, from simple applications to sophisticated, multi-tasking, observing and data analysis systems. A typical ADAM system consists of a number of tasks which communicate with each other following well defined protocols. (On VMS, each task is a separate VMS process.) Tasks are written using a standard set of subroutine libraries which provide the ADAM facilities. ADAM is often termed a *software environment* as, in a completely ADAM system, it is what the user's application code 'sees' around itself.

ADAM was originally developed by the Royal Greenwich Observatory to run on the Perkin-Elmer computers of the INT and JKT on La Palma, for instrument control on these telescopes. The Royal Observatory Edinburgh adapted ADAM to run on VAX/VMS systems to provide the instrument control environment for UKIRT. In doing so they incorporated most features of the Starlink Software Environment (SSE) so that SSE programs were equivalent to ADAM A-tasks. VAX ADAM has now been adopted as the standard instrument control environment for the AAT, the WHT at La Palma and the JCMT on Mauna Kea as well as UKIRT. ADAM has also been adopted by Starlink as its standard environment for data reduction. Responsibility for support of ADAM now rests with the ADAM Support Group, which is part of the Starlink project.

There are several references in the following chapters to Starlink User Notes (SUNs), Starlink System Notes (SSNs) and Starlink Guides (SGs). On Starlink systems they will be found in a directory with logical name DOCSDIR. Your site manager should be able to provide hardcopies.

8.2 The Role of the Command Language

The command language was originally conceived as playing a key role in the ADAM system by providing the only user interface to the system. However, other user interfaces to ADAM have been developed, so users of ADAM systems will not necessarily find themselves using ICL (or its predecessor ADAMCL) when working with an ADAM system.

When using ADAM for data reduction the command language will probably be the standard means of running ADAM. When ADAM is used for on line instrument control the command language can be used, but this is generally done only in the testing phase of the instrument. In fully developed systems the instrument will probably be controlled through a user interface which makes more sophisticated use of the terminal. There are two such systems in current use.

At UKIRT, the instrument control software is using the screen management system (SMS). With SMS the user is presented with menus from which selections are made using the cursor keys. The SMS menu selections actually result in command language code being executed, but the user does not normally interact with the command language directly.

At the AAT, the user interface for instrument control is currently by means of ADAM tasks known as U-tasks. The user of a U-task sees a screen divided into a fixed region in which status information on the

instrument or observing process is displayed, and scrolling regions for message output and command input. The command language is not normally involved, though it is always possible to use the command language to control a U-task, which is equivalent in this respect to any other ADAM task.

Chapter 9
Using ADAM for Data Reduction

9.1 A-tasks

When ADAM is used for data reduction, ICL is used to communicate with one or more A-tasks or application tasks which contain the data reduction programs. Each of these tasks can be associated with an ICL command by means of the DEFINE command. When the associated ICL command is issued the A-task will be loaded and executed. If the same ICL command is issued a second time the task will not normally need to be reloaded so that execution will be faster. However, if too many tasks are loaded one of these tasks may have to be killed before another can be loaded. The maximum number of tasks allowed to be loaded at one time has a default value of three, but may be adjusted, on a system basis, by the system logical name ADAM_CACHLIMIT. The least recently used task is always selected for killing when the limit is exceeded. This process is known as task caching and such tasks are referred to as cached tasks.

9.2 Monoliths

To avoid frequent loading and killing of tasks, large data reduction packages are normally organized as *Monoliths* or M-tasks. A monolith is a single task which contains a large number of independent programs which would otherwise be separate A-tasks. Thus a monolith will have many commands associated with it, whereas an A-task only has a single command. The monolith will be loaded when the first command is issued and then any of the other monolith commands may be issued without requiring the task to be reloaded.

9.3 Running KAPPA

An example of an ADAM monolith is the Starlink Kernel Applications Package (KAPPA). KAPPA is described in more detail in the Starlink document SUN/95. To run KAPPA type:

```
$ ADAM KAPPA
```

This is actually a convenient way of combining the commands[1]:

```
$ ADAMSTART
$ ICL
ICL> KAPPA
```

[1] In fact, any ICL command may be specified in place of KAPPA in the ADAM command.

To run any ADAM task from ICL it is necessary to have obeyed the command ADAMSTART in DCL before running ICL. If you use ADAM tasks frequently, you may include this command in your LOGIN.COM file. The ADAM procedure will not obey ADAMSTART again if it has already been obeyed, although it wouldn't matter if it did.

When the command KAPPA is obeyed in ICL, you will see

```
 Help key KAPPA redefined

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

    Type HELP KAPPA or KAPHELP for KAPPA help
ICL>
```

The KAPPA command causes all the individual commands of the KAPPA monolith to be defined and then outputs its initialization message. It does not cause the KAPPA monolith itself to be loaded. This occurs when the first KAPPA command is issued.

The simplest way of using KAPPA is to type just the command names. You will then be prompted for all the required parameters. For example, to use the ADD command which adds two images just type ADD and the following dialogue will result

```
ICL> ADD
Loading KAPPA_DIR:KAPPA into 0591KAPPA
IN1 - First input NDF> IM1
IN2 - Second input NDF> IM2
OUT - Output NDF> SUM
ICL>
```

The above command causes the images contained in HDS data files IM1.SDF and IM2.SDF to be added and the result placed in a new file called SUM.SDF.

Since this is the first KAPPA command the monolith needs to be loaded, and a loading message[2] is output which tells us the name of the task being created, in this case 0591KAPPA[3].

Each parameter prompt consists of three parts, the name of the parameter (*e.g.* IN1), a brief description (First input NDF[4]), and in some cases a *suggested* value enclosed between / characters.

When responding to a parameter prompt the user has several options.

- Just hit the return key. The suggested value will be used.
- Type in a new value.
- Hit the TAB key. This causes the suggested value to be placed in the input buffer. It can then be edited by the normal command line editing keys.
- Enter ! — the NULL value. The effect of doing this will depend upon the application but frequently causes termination.
- Enter !! — by convention, this causes the program to be aborted.

[2]These loading messages can be turned off by the SET NOMESSAGES command, if desired

[3]The name is prefixed with a number (*e.g.* 0591) to make a unique process name, so that there is no conflict between your version of KAPPA, and another version run at the same time by another user

[4] 'NDF' stands for *Extensible N-Dimensional Data Format*. For more information see Section 10.7.

- Enter ? or ?? — this causes the help system to be entered, giving further information on the parameter and/or the application (assuming that such text has been provided by the application implementor[5]).

If the parameters of a command are known they can be placed on the command line itself. Thus the above example would become:

```
ICL> ADD IM1 IM2 SUM
```

In this example it is important to get the parameters in the right order. If we are not sure about the order of the parameters, but know their names we can specify the parameters by name on the command line.

```
ICL> ADD OUT=SUM IN1=IM1 IN2=IM2 TITLE='Sum of 2 Images'
```

TITLE is an example of a parameter which can only be specified using the '*name*=' syntax as it has not been allocated a position in the order of parameters. If TITLE is not specified, a default value will be used.

9.4 ADAM Parameter Format

ADAM task parameters can be of a number of different types as follows:

- Numbers — These can be Integer, Real or Double Precision and are entered in the usual format (*e.g.* as in FORTRAN)
- Strings — These are represented in the usual ICL formats. Quotation marks may be omitted where there is no ambiguity. In the above example they were necessary on 'Sum of 2 images' when used on the command line as the spaces would otherwise make it appear to be four different parameters. However, they would not be needed for the same string in response to a single parameter prompt.
- Logical values — These can be represented by the words TRUE, FALSE, YES, NO, T, F, Y or N, regardless of case.
- Arrays — arrays of any of the above items 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.* The outer brackets may be omitted when responding to a prompt.
- Names — These are the names of HDS (hierarchical data system) objects or files, or the names of devices (graphics devices, tape drives *etc.*). Normally these names can be typed directly as MTA0, ARGS etc, but there are a few cases of ambiguity. These cases can be resolved by prefixing the name with an @ character. For example, if it is required to specify an HDS container file with a name different from that of the object it contains, or with a specific generation number, the file specification can be enclosed in quotes (*e.g.* `"data.sdf;3"`). However, this would be interpreted as a character string unless prefixed with @. Suggested values in prompts are always displayed prefixed with @.

[5]For a detailed description of the parameter help system, see SUN/115.

9.5 KAPPA from Procedures

ICL becomes particularly useful when a series of KAPPA operations are to be performed on a sequence of files. We can then use an ICL procedure for this purpose. In such cases we will probably want to use an ICL variable or expression to specify at least one of the parameter names. This is perfectly acceptable provided the expression is placed in parentheses so that it will be evaluated and not treated as a string. A frequent occurrence is that we want to process a sequence of files which have names such as RUN1, RUN2, RUN3 *etc.* ICL therefore provides a function SNAME to generate such sequential names [6]. It has the form:

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

and produces a name which is the concatenation of the string with the integer n. An optional 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.

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

The latter example being the format of name produced directly by the IPCS observing software at the AAT. Using this way of specifying the name it is easy to write an ICL procedure to add a whole series of images together using the KAPPA ADD command.

```
PROC KADD

{   Add images RUN1 to RUN20 to form SUM  }

   ADD RUN1 RUN2 SUM TITLE='Sum of 2 images'
   LOOP FOR I=3 TO 20
     TITLE = '''Sum of ' & I:2 & ' images'''
     ADD  SUM  (SNAME('RUN',I))  SUM  TITLE=(TITLE)
   END LOOP

END PROC
```

9.6 Error Reports

In the event of an error occurring in the task, error reports will be displayed. For example:

```
ICL> ADD
Loading KAPPA_DIR:KAPPA into 0591KAPPA
IN1 - First input NDF > !
!! Null NDF structure specified for the IN1 parameter
!  ADD: Error adding two NDF data structures
ADAMERR   %PAR, Null parameter value
```

The first two messages are issued by the task. Such messages will usually indicate which ADAM subsystem or routine generated them by a prefix (ADD: in the example).

The message starting 'ADAMERR' is issued by ICL on receiving a termination message containing a 'bad' status. ADAMERR is the name of an ICL exception – the %PAR following ADAMERR tells us that the PAR subsystem within ADAM generated the bad status value.

[6] Note that for reasons explained in Section 10.5, strings defining names to be used as ADAM task parameters are generally prefixed with @.

Chapter 10
Writing ADAM tasks

10.1 Introduction

It is easy to write your own ADAM A-tasks which can be run from ICL in the same way as the KAPPA programs. This is the easiest way of allowing ICL to run your own FORTRAN programs[1]. A detailed discussion of writing A-tasks can be found in SG/4 and SUN/101.

Here is a simple example of an ADAM A-task. Its source code consists of the FORTRAN source file MYTASK.FOR containing the following:

```
SUBROUTINE MYTASK(STATUS)
IMPLICIT NONE
INTEGER STATUS

CALL MSG_OUT(' ','Hello',STATUS)

END
```

The example task consisted essentially of just one statement — the call of MSG_OUT. This routine is part of a subroutine library which is provided for outputting information to the terminal. The MSG_ routines should always be used for terminal output from ADAM tasks. FORTRAN PRINT and WRITE statements must not be used for this purpose.

Every ADAM task also has an interface file. The interface file contains information on the parameters of the task. Our task doesn't have any parameters so its interface file is fairly simple. It consists of the following in the file MYTASK.IFL.

```
INTERFACE MYTASK
ENDINTERFACE
```

10.2 Compiling and Linking

Before compiling and linking ADAM tasks we need to run the command ADAM_DEV which sets up appropriate definitions. This in turn requires that the command ADAMSTART has been run.

```
$ ADAMSTART
$ ADAM_DEV
```

The task can then be compiled and linked using the following commands:

[1] The alternative way is by using the DEFUSER command

```
$ FOR MYTASK
$ ALINK MYTASK
```

To run the task start up ICL and define a command to run the task using the ICL DEFINE command.

```
ICL> DEFINE  TEST  MYTASK
ICL> TEST
Loading MYTASK into 03BCMYTASK
Hello
ICL> TEST
Hello
ICL>
```

The DEFINE command defines the command TEST to run the A-task MYTASK. For this to work, MYTASK has to be in the default directory. If it were somewhere else a directory specification could be included on MYTASK in the DEFINE command.

TEST then causes the task to be loaded and executed. Typing TEST again causes it to be executed a second time, but this time it doesn't have to be loaded.

10.3 Tasks with Parameters

The following is an example of a task with a parameter. This task calculates the square of a number and outputs its value on the terminal.

```
  SUBROUTINE SQUARE(STATUS)
  INTEGER STATUS
  REAL R,RR
  CALL PAR_GET0R('VALUE',R,STATUS)
  RR = R*R
  CALL MSG_SETR('RVAL',R)
  CALL MSG_SETR('RSQUARED',RR)
  CALL MSG_OUT(' ','The Square of ^RVAL is ^RSQUARED',STATUS)
  END
```

This program uses the subroutine PAR_GET0R to get the value of the parameter VALUE (there are similar routines for other types). Output of the numbers is done using the routine MSG_SETR to give values to the tokens RVAL and RSQUARED which are then inserted into the MSG_OUT output string using the `^RVAL` notation. The interface file for this example is given below.

```
INTERFACE SQUARE

  PARAMETER VALUE
    TYPE _REAL
    POSITION 1
    VPATH PROMPT
    PPATH CURRENT
    PROMPT 'Number to be squared'
  ENDPARAMETER

ENDINTERFACE
```

The interface file has an entry for the parameter VALUE. The TYPE field specifies the type of the parameter. The underscore prefix on '_REAL' identifies it as a *primitive* type (*i.e.* a simple number or string, rather than an HDS structure or Device name). The position field specifies the position that the parameter is expected in if it appears on the command line.

The VPATH entry specifies how the parameter value is to be obtained if it not found on the command line. In this case it is to be prompted for. The PPATH entry specifies how the default value that appears in the parameter prompt is to be obtained. In this case the CURRENT value (*i.e.* the value the parameter had at the end of the last execution of the command) is used. The PROMPT field gives the prompt string to be used.

Interface files are described in more detail in SUN/115.

To run this example we would compile and link it as described above and then use the following ICL commands:

```
ICL> DEFINE SQUARE SQUARE
ICL> SQUARE 12
Loading SQUARE into 03BCSQUARE
The Square of 12 is 144
ICL> SQUARE (SQRT(3))
The Square of 1.73205 is 3
ICL> SQUARE
VALUE - Number to be squared /0.173205E+01/ > 7
The Square of 7 is 49
ICL>
```

10.4 STATUS and error handling

Most ADAM subroutines have an integer parameter called STATUS. STATUS has a success value (SAI__OK) which each routine will return if it completes successfully. If the routine fails for some reason it will return an error code indicating the nature of the error. An ADAM A-task routine (such as SQUARE) will be called with a STATUS of SAI__OK. If it returns a bad status value to its caller this will result in an appropriate message being output.

There is a further important feature of the status convention. If an ADAM routine is called with its STATUS argument having an error value on input, then the routine will do nothing and will return immediately. This feature means that it is usually not necessary to check STATUS after each routine is called. A series of ADAM routines can be called with the 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 STATUS value is then returned by the A-task main routine to its caller an error message will result. Thus the error will be correctly processed with no special code being added to check for errors.

It is important, however, to take care that code that does not consist of calls to ADAM routines does not get executed after an error has occurred. For this reason our SQUARE example would be better written as:

```
SUBROUTINE SQUARE(STATUS)
INTEGER STATUS
INCLUDE 'SAE_PAR'  ! This provides the ADAM status codes
REAL R,RR
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)
   ENDIF
   END
```

This ensures that if the STATUS from PAR_GET0R is bad the rest of the routine is not executed with an undefined value of R. It is actually not necessary to include MSG_OUT in the IF block as this would not execute if STATUS was bad.

More sophisticated error handling can be provided by using routines in the ERR_ package. These facilities are fully described in SUN/104.

10.5 Returning values to ICL

We have already seen how ICL can be used to supply values for ADAM task parameters. It is also possible for ADAM tasks to return values to ICL. The following modified version of SQUARE does not output its result on the terminal, but returns it to the parameter VALUE using a call to the routine PAR_PUT0R which is analogous to PAR_GET0R.

```
SUBROUTINE SQUARE(STATUS)
INTEGER STATUS
REAL R,RR
CALL PAR_GET0R('VALUE',R,STATUS)
IF (STATUS .EQ. SAI__OK) THEN
   RR = R*R
   CALL PAR_PUT0R('VALUE',RR,STATUS)
ENDIF
END
```

We could run this from ICL as follows (having done a DEFINE SQUARE SQUARE) to define the command:

```
ICL> X=5
ICL> SQUARE (X)
ICL> =X
25
ICL>
```

In order for the ADAM task to return a value to ICL we must use a variable for the parameter and place it on the command line. The variable name must be placed in parentheses, then the name of a temporary HDS object is substituted by ICL.

A modification of this scheme is needed with character variables to allow the case where the contents 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 @.

10.6 Graphics with ADAM

All graphics in ADAM is based on the use of the GKS graphics system. Most users, however, will not use the GKS routines directly but will use a higher level package such as SGS, NCAR or PGPLOT. The following example uses SGS to draw a circle on a selected graphics device. (SGS is described in SUN/85, and its use within ADAM in SUN/113.)

```
      SUBROUTINE CIRCLE(STATUS)
      IMPLICIT NONE
      INTEGER STATUS
      REAL RADIUS          ! Radius of circle
      INTEGER ZONE         ! SGS Zone
      INCLUDE 'SAE_PAR'    ! Adam Constants

*   Get the radius of the circle
      CALL PAR_GETOR('RADIUS',RADIUS,STATUS)

*   Get the graphics device, and open SGS
      CALL SGS_ASSOC('DEVICE','WRITE',ZONE,STATUS)

*   Draw the circle
      IF (STATUS .EQ. SAI__OK) THEN
          CALL SGS_CIRCL(0.5,0.5,RADIUS)
      ENDIF

*   Close the graphics workstation
*   and cancel the parameter
      CALL SGS_CANCL(ZONE,STATUS)

*   Close down SGS
      CALL SGS_DEACT(STATUS)

      END
```

The interface file for this example is as follows:

```
INTERFACE CIRCLE

   PARAMETER RADIUS
      TYPE _REAL
      POSITION 1
      VPATH PROMPT
      PPATH CURRENT
      PROMPT 'Radius of Circle'
   ENDPARAMETER

   PARAMETER DEVICE
      PTYPE DEVICE
      POSITION 2
      VPATH PROMPT
      PPATH CURRENT
      PROMPT 'Graphics Device'
   ENDPARAMETER

ENDINTERFACE
```

When SGS is used from ADAM no calls to SGS_OPEN and SGS_CLOSE are made. Instead the routines SGS_ASSOC and SGS_ANNUL are used. SGS_ASSOC makes the association between an ADAM parameter (DEVICE) and an SGS zone whose zone identifier is returned to the program in the ZONE parameter. When SGS plotting is finished SGS_ANNUL is called and given the same ZONE value. SGS_ASSOC has an additional parameter, the access mode, which has possible values 'READ', 'WRITE' and 'UPDATE'.

There are a number of similar pairs of _ASSOC and _CANCL routines in ADAM which work in similar ways. MAG_ASSOC and MAG_CANCL are used to handle magnetic tape devices, FIO_ASSOC and FIO_CANCL to handle file I/O *etc.* Many of them also have an _ANNUL subroutine which frees the associated resource but does not cancel the associated ADAM parameter.

To run CIRCLE from ICL we could either use:

```
ICL> CIRCLE  0.3  ARGS1
```

or let it prompt for the parameters:

```
ICL> CIRCLE
RADIUS - Radius of Circle /0.300000E+00/ > 0.2
DEVICE - Graphics Device /@ARGS1/ > PRINTRONIX
ICL> $ PRINT/NOFEED PRINTRONIX.BIT
ICL>
```

In the latter case the default values of the parameters are the values from the previous time (a result of using PPATH CURRENT). With a hard copy graphics device such as PRINTRONIX, a file is created which must then be sent to the device with a DCL PRINT command.

10.7 Accessing Data

The basic means of storing and accessing data for ADAM is the Hierarchical Data System (HDS). SUN/92 describes this system and the DAT_ package of routines that are used to access data in this form. In HDS a data file contains a number of named components which can either be primitive items (numbers, character strings or arrays) or can themselves be structures containing further components.

To simplify the exchange of data between different applications packages, Starlink have released a set of standards for representing data within HDS. This is the *Extensible N-Dimensional Data Format* (NDF) – it is described in SGP/38. A library of subroutines (the NDF library, described in SUN/33) is provided for accessing these standard structures and will generally be used when writing applications.

Below is a simple example that calculates the mean value of the data in an NDF. For such data files the data will be found in a component of the file, called .DATA.

```
      SUBROUTINE MEAN(STATUS)
      IMPLICIT NONE
      INTEGER STATUS
      INCLUDE 'SAE_PAR'
      INTEGER NELM                      ! Number of Data elements
      CHARACTER*(DAT__SZLOC) LOC        ! HDS locator
      INTEGER PNTR                      ! Pointer to Data
      REAL MN                           ! Mean value of data

*  Start an NDF context
      CALL NDF_BEGIN
```

```
* Get locator to parameter
      CALL NDF_ASSOC('INPUT','READ',LOC,STATUS)

* Map the data array
      CALL NDF_MAP(LOC,'DATA','_REAL','READ',PNTR,NELM,STATUS)

* If everything OK calculate mean value of data array and output it
      IF (STATUS .EQ. SAI__OK) THEN
          CALL MEAN_SUB(NELM,%VAL(PNTR),MN)
          CALL MSG_SETR('MEAN',MN)
          CALL MSG_OUT(' ','Mean Value of Array is ^MEAN',STATUS)
      ENDIF

* End the NDF context
      CALL NDF_END(STATUS)

      END


      SUBROUTINE MEAN_SUB(NELM,ARRAY,MEAN)

*  Subroutine to calculate the mean value of the array

      IMPLICIT NONE
      INTEGER NELM
      REAL ARRAY(NELM)
      REAL MEAN
      INTEGER I

      MEAN = 0.0
      DO I=1,NELM
         MEAN = MEAN + ARRAY(I)
      ENDDO
      MEAN = MEAN / NELM
      END
```

There are a number of points to note about this example:

- The routine NDF_ASSOC is analogous to SGS_ASSOC as used in the previous example. It returns an NDF identifier associated with an ADAM parameter (INPUT).
- The routine NDF_MAP returns a pointer (PNTR) to the data array. PNTR is an integer variable which contains the memory address of the data. Mapping is used in applications of this type as an alternative to reading the data into an array. The advantage of mapping is that we don't have to make a guess at the maximum size of array we expect, in order to know how big an array to declare. NDF_MAP will return a pointer to a real array. If the original data is of a different type it will be converted to real.
- The pointer concept is not supported in standard FORTRAN, but can be made use of in VAX FORTRAN by passing the pointer to a subroutine using the %VAL(PNTR) construct. In the subroutine (MEAN_SUB in this case) the corresponding parameter can then be treated as if we had passed the actual array.
- The routine NDF_END is called to end the current NDF context and unmap the mapped data array. For more information on NDF contexts see SUN/33.

The interface file for this example could be as follows:

```
INTERFACE MEAN

   PARAMETER INPUT
      TYPE NDF
      POSITION 1
      VPATH PROMPT
      PPATH CURRENT
      PROMPT 'NDF to calculate Mean Value from'
   ENDPARAMETER

ENDINTERFACE
```

The above example shows how to handle the case where an NDF file is used for input. Where output to an NDF file is involved it is usually necessary to create a new HDS file when the application runs. This can be achieved using routine NDF_CREAT or NDF_CREP. For details of their usage, see SUN/33.

Chapter 11
ADAM as a Data Acquisition Environment

11.1 Instrumentation Tasks

When used for data acquisition, ICL is used to control one or more 'instrumentation tasks' (I-tasks). A I-task is used to control an individual instrument or other hardware component of the system. An I-task can respond to a number of different commands (referred to as *Actions*) rather than the single command of an A-task; they also have the ability to perform two or more actions concurrently.

Multiple I-tasks may be involved when several instruments are used in combination and sometimes it is convenient to have another I-task controlling the individual instrument I-tasks. In this case, ICL would be used to control the controlling task which would relay to ICL any messages or prompts from the subsidiary I-tasks intended for the operator.

Instrumentation tasks are fully described in SUN/134.

11.2 D-tasks and C-tasks

Instrumentation tasks combine the functions of Device (D) task and Control (C or CD) tasks which were used with ADAM V1. These old-style tasks are now considered 'unfashionable' but they continue to work with ICL and it will be some time before they are replaced in all systems.

11.3 Task Loading

In a data acquisition situation the task caching scheme described in the previous section is usually inappropriate. We don't want tasks to be killed when the caching limit is exceeded, and we may not want a task to be killed when ICL exits (as happens with cached tasks). Also the unique process names created for cached tasks are undesirable as many different tasks may want to communicate with a given task, and therefore need to know its name.

Therefore I-tasks are normally loaded as uncached tasks, and this is achieved by explicitly loading them using one of the three ICL load commands.

ALOAD — Loads a task into a subprocess without waiting for completion.

LOADW — Loads a task into a subprocess and waits for completion.

LOADD — Loads a task into a detached process and waits for completion.

11.4 Killing Tasks

Uncached tasks remain loaded until explicitly killed, or until the creating process is logged out. They remain loaded when ICL exits. Thus it is possible to use ICL to load a task, then exit from ICL, and subsequently communicate with the task from a second invocation of ICL (which might be started on a different terminal).

Tasks are killed using the KILL or KILLW commands.

KILL — Kills a task without waiting for completion

KILLW — Kills a task and waits for completion

11.5 The ADAM message system

Communication between ADAM tasks, and between ICL and tasks, makes use of the ADAM message system. The message system is involved in the communication between ICL and A-tasks described in the last chapter (§10), but in this case its details are largely hidden from the user. In the data acquisition case the use of the message system by the command language is usually more explicit.

ICL can send four types of messages to tasks, which are distinguished by a *context* which is one of GET, SET, OBEY or CANCEL.

The ICL command to send a message to a task is SEND.

11.6 An Example

As an example we will consider a I-task called PHOTOM which controls a simple optical photometer. The photometer includes a filter wheel with five positions.

11.6.1 The I-task Interface File

The I-task has a parameter FILTER_DEMAND which specifies the required filter and an action FILTER which moves the filter wheel to the required position. These are specified in the interface file for the task as follows:

```
PARAMETER FILTER_DEMAND
  TYPE '_CHAR'
  VPATH 'INTERNAL'
  IN 'U','B','V','R','I'
ENDPARAMETER

ACTION FILTER
  OBEY
    NEEDS FILTER_DEMAND
  ENDOBEY
  CANCEL
  ENDCANCEL
ENDACTION
```

Many I-task parameters have VPATH specified as 'INTERNAL' – this means that the parameter value is stored in memory for speed of access. The IN field specifies the valid values for the parameter (it is also possible to use a RANGE specification to restrict the values of a parameter).

The action entry specifies that the action FILTER may be obeyed and cancelled, and that they OBEY action needs the parameter FILTER_DEMAND which may be specified as the first (only) parameter on the command line.

11.6.2 Running PHOTOM

The I-task could be used as follows:

```
ICL> LOADW PHOTOM
Loading PHOTOM into PHOTOM
ICL> SEND PHOTOM SET FILTER_DEMAND V
ICL> SEND PHOTOM GET FILTER_DEMAND
V
ICL> SEND PHOTOM OBEY FILTER
ICL>
```

The last command starts the FILTER action but does not wait for it to complete. ICL can accept and execute other commands while the filter wheel is moving to its position. When the action finally completes the I-task returns a completion message to ICL which will be output on the terminal.

```
ICL>
"obey" FILTER - action complete
```

11.6.3 Supplying Parameters in the Obey Message

A parameter in the NEEDS list for an action may be supplied along with the OBEY message rather than set independently with a SET message. Thus we can use:

```
ICL> SEND PHOTOM OBEY FILTER B
```

11.6.4 Cancelling Actions

The CANCEL context enables us to cancel the FILTER action before it has completed by issuing the following command:

```
ICL> SEND PHOTOM CANCEL FILTER
```

11.6.5 Missing Parameters

If we try to start the FILTER action without specifying a value for the FILTER_DEMAND parameter (either by SEND SET or by specifying it on the command line), the parameter system will attempt to get a value in the same way that it does for A-tasks (*but note that 'INTERNAL' parameters will not be prompted for – a pseudo VPATH of 'DYNAMIC,DEFAULT' is used and there is no implied PROMPT at the end.*) Therefore, as there is no DEFAULT value specified for FILTER_DEMAND, and assuming that no dynamic default value is specified by the program, the null response is assumed.

```
ICL> SEND PHOTOM OBEY FILTER
Bad status returned from task, meaning is :-
ICL>
%PAR-I-NULL, Null parameter value
ICL>
```

Note that old-style D-tasks always had parameters defined as 'INTERNAL' and could not prompt for missing parameters anyway.

11.6.6 The GET command

A SEND PHOTOM GET command causes the parameter value to be output on the terminal. It is often more useful to put the value into an ICL variable. This can be accomplished using the GET command.

```
ICL> GET PHOTOM FILTER_DEMAND (F)
ICL> =F
B
ICL>
```

11.6.7 The OBEYW command

It is often more convenient to let ICL wait for an action to complete rather than have it proceed concurrently with other ICL commands. The OBEYW command is used for this purpose.

```
ICL> OBEYW PHOTOM FILTER I
ICL>
```

The OBEYW command waits until the action has completed before the ICL> prompt reappears and more commands may be entered.

11.6.8 Multiple Concurrent Actions

I-tasks have the ability to perform more than one action concurrently. ICL includes two commands which enable actions to be performed concurrently while waiting for all of the actions to complete. Suppose our PHOTOM I-task has another action APERTURE which moves the aperture wheel to a required position. The following is a procedure which would allow the filter and aperture wheels to be moved simultaneously.

```
    PROC SETUP  F  A

{  move filter to position F and aperture to position A  }

{  Start both actions  }

      STARTOBEY (P1) (M1) PHOTOM FILTER (F)
      STARTOBEY (P2) (M2) PHOTOM APERTURE (A)

{  Wait for the actions to complete  }

      ENDOBEY (P1) (M1)
      ENDOBEY (P2) (M2)

    END PROC
```

STARTOBEY starts an action and returns the *path* and *message identifier*, which together uniquely identify a given invocation of an action, into two ICL variables. ENDOBEY can then be used to wait for the completion of such an action. The above procedure will therefore exit only when both actions are complete.

Appendix A
ICL Functions

Where the type is given as 'Real or Integer' the type of the result is the same as the type of the argument.

Name	Type	Definition
ABS(X)	Real or Integer	$\mid x \mid$
ASIN(X)	Real	$\sin^{-1} x$ *where* $-1 \leq x \leq 1$ $-\pi/2 \leq result \leq \pi/2$
ASIND(X)	Real	$\sin^{-1} x$ *where* $-1 \leq x \leq 1$ $-90 \leq result \leq 90$
ACOS(X)	Real	$\cos^{-1} x$ *where* $-1 \leq x \leq 1$ $0 \leq result \leq \pi$
ACOSD(X)	Real	$\cos^{-1} x$ *where* $-1 \leq x \leq 1$ $0 \leq result \leq 180$
ATAN(X)	Real	$\tan^{-1} x$ $-\pi/2 \leq result \leq \pi/2$
ATAND(X)	Real	$\tan^{-1} x$ $-90 \leq result \leq 90$
ATAN2(X1,X2)	Real	$\tan^{-1}(x_1/x_2)$ $-\pi < result \leq \pi$ according to quadrant in which point (x_2, x_1) lies

Name	Type	Definition
ATAN2D(X1,X2)	Real	$\tan^{-1}(x_1/x_2)$ $-180 < result \leq 180$ according to quadrant in which point (x_2, x_1) lies
BIN(I,n,m)	String	integer I formatted in binary into an n char string with m significant digits
CHAR(I)	String	Character whose ASCII value is I
COS(X)	Real	$\cos x$ (x in radians)
COSD(X)	Real	$\cos x$ (x in degrees)
COSH(X)	Real	$\cosh x$
DATE()	String	The Current Date
DEC(I,n,m)	String	integer I formatted in decimal into an n char string with m significant digits
DECL(S)	Real	A Declination in Radians from a string in degrees, minutes, seconds
DEC2S(R,NDP,SEP)	String	String in DMS from Dec in radians. NDP decimal places on seconds. Character SEP separates fields
DIM(X1,X2)	Real or Integer	Positive difference $x_1 - min(x_1, x_2)$
ELEMENT(I,DELIM,S)	String	Ith element of delimited string, I ≥ 0
EXP(X)	Real	e^x
FILE_EXISTS(S)	Logical	TRUE if file S exists
FLOAT(I)	Real	Integer I converted to real

Name	Type	Definition
HEX(I,n,m)	String	integer I formatted in hexadecimal into an n char string with m significant digits
IAND(I1,I2)	Integer	Bitwise AND of two integers
ICHAR(S)	Integer	ASCII value of first character of String
IEOR(I1,I2)	Integer	Bitwise exclusive OR of two integers
IFIX(X)	Integer	Real to Integer by Truncation Equivalent to INT
INDEX(S1,S2)	Integer	Position of first occurrence of Pattern S2 in string S1 (Zero if not found)
INKEY()	Integer	Key value of last trapped key pressed Only works in screen mode
INOT(I)	Integer	Bitwise Complement of Integer
INT(X)	Integer	Real to Integer by Truncation
INTEGER(X)	Integer	The value of X converted to an integer Equivalent to NINT
IOR(I1,I2)	Integer	Bitwise OR of two integers
KEYVAL(S)	Integer	Value of key with name S
LEN(S)	Integer	Length of String S
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)

Name	Type	Definition
LOG10(X)	Real	$\log_{10} x$
LOGICAL(X)	Logical	The value of X converted to logical
MAX(X1,X2, ...)	Real or Integer	Maximum of two or more arguments
MIN(X1,X2, ...)	Real or Integer	Minimum of two or more arguments
MOD(X1,X2)	Real or Integer	$x_1 - int(x_1/x_2) \times x_2$
NINT(X)	Integer	Nearest integer to X
OCT(I,n,m)	String	integer I formatted in octal into an n char string with m significant digits
OK(stat)	Logical	True if VMS Status OK
RA(S)	Real	Right Ascension in Radians from a string in hours, minutes, seconds
RA2S(R,NDP,SEP)	String	String in HMS from RA in radians. NDP decimal places on seconds. Character SEP separates fields
RANDOM(I)	Real	Random number between 0 and 1 from seed I I must be a variable
REAL(X)	Real	The value of X converted to real
SIGN(X1,X2)	Real or Integer	x_1 with sign of x_2
SIN(X)	Real	$\sin x$ (x in radians)
SIND(X)	Real	$\sin x$ (x in degrees)
SINH(X)	Real	$\sinh x$

Name	Type	Definition
STRING(X)	Logical	The value of X converted to a string
SQRT(X)	Real	$\sqrt{x}$
SUBSTR(S,n,m)	String	Substring of S beginning at n (≥ 1) of length m
TAN(X)	Real	tan x (x in radians)
TAND(X)	Real	tan x (x in degrees)
TANH(X)	Real	tanh x
TIME()	String	The current time
TYPE(X)	String	The Type of Variable X 'REAL', 'INTEGER', 'LOGICAL', 'STRING' or 'UNDEFINED'
UNDEFINED(X)	Logical	TRUE if X is undefined
UPCASE(S)	String	String S converted to Upper case
VARIABLE(proc,X)	Any	Returns value of variable X of procedure proc

Appendix B
ICL Commands

B.1 ALLOC

ALLOC dev [(actdev)] [(status)]

Allocate a device:

dev - The name of the device to be allocated This may be a generic name in which case the first available device will be allocated.

actdev - (optional) A variable in which the name of the device actually allocated will be returned.

status - (optional) A variable in which the status of the allocate request will be returned. The success of the request may be tested using the function OK(status).

ALLOC should be used in preference to the $ ALLOCATE command when the device is to be used by an ICL procedure. For example to allocate a tape drive used by a FIGARO input command.

ALLOC may be abbreviated to ALL

e.g.

```
ALLOC MT (DEVICE) (STATUS)
```

B.2 ALOAD

ALOAD exename [taskname]

Load an ADAM task into a subprocess:

exename - The name of the executable image file to be used for the task.

taskname - (optional) The name of the task to be created. If omitted it defaults to the file name part of exename.

B.3 APPEND

APPEND intname [filename]

Open an existing text file for output. Output will be appended to the existing text.

intname - The internal name by which the file will be known within ICL.

filename - The VMS file name of the file. If omitted the file name will be intname.DAT.

B.4 CHECKTASK

CHECKTASK taskname (loaded)

Check whether an ADAM task is currently loaded.

taskname - The name of the task.

loaded - An ICL variable which will be set to the logical value TRUE if the task is loaded, FALSE otherwise.

B.5 CLEAR

CLEAR first last

In screen mode the range of lines on the screen between *first* and *last* are cleared.

B.6 CLOSE

CLOSE intname

Close a text file previously opened with CREATE, OPEN or APPEND.

intname - The internal name of the file.

B.7 CREATE

CREATE intname [filename]

Create a text file and open it for output.

intname - The internal name by which the file will be known within ICL.

filename - The VMS file name of the file. If omitted the file name will be intname.DAT.

B.8 CREATEGLOBAL

CREATEGLOBAL parname type

Create an ADAM global parameter.

parname - The name of the global parameter to be created. If a parameter of this name already exists it will be deleted.

type - The type of the parameter to be created. Must be one of _REAL, _INTEGER, _DOUBLE, _LOGICAL, or _CHAR*n (where n is an integer).

B.9 $ (DCL)

$ command

The parameters of the $ command are concatenated to form a command which is executed by the underlying operating system's command language. 'DCL' may be used as an alternative to '$'.

If the underlying command language is DCL, the specified command is obeyed within a permanent DCL subprocess which is created on the first call to DCL.

Because the commands are executed in the DCL subprocess rather than in the process running ICL, some commands may not have the expected effect. *e.g.* $ SET DEF will set the default directory for the subprocess but not for the process running ICL. Use the ICL DEFAULT (DEF) command instead. Furthermore, $ ALLOCATE will allocate a device to the subprocess and not to the process running ICL. Use the ICL command ALLOC for this purpose.

Control-C may be used to abort a DCL command running in the subprocess.

B.10 DEALLOC

DEALLOC dev

Deallocate a device previously allocated using the ICL ALLOC command.

DEALLOC may be abbreviated to DEALL.

B.11 DEFAULT

DEFAULT directory

Set the default directory for both the process running ICL, and for the DCL subprocess. The directory may be specified in any of the forms accepted by the DCL SET DEF command. DEFAULT with no parameter displays the current default directory.

DEFAULT should be used in preference to DCL SET DEF which will only set the default directory for the DCL subprocess.

DEFAULT may be abbreviated to DEF.

e.g.

```
DEF [-]
DEF DISK$DATA:[ABC.DATA]
```

B.12 DEFHELP

DEFHELP name help_library [topic...]

Allows the ICL HELP command to be able to access information in other help libraries.

name - The Help name to be defined.

help_library - The Help library to be used.

topic... - An optional list of keywords specifying the path within *help_library* to the required information. If omitted, *topic* defaults to *name*.

After *name* has been defined in this way

```
ICL> HELP name
```

will return the *topic* information in *help_library*.

B.13 DEFINE

DEFINE command taskname [action]

Define a command which issues an OBEYW to an ADAM task

command - The command to be defined. An abbreviation may be specified in the form COM(MAND) where COM is the minimum acceptable abbreviation.

taskname - The task which will execute the OBEYW.

action - (optional) The action to be executed. If omitted it defaults to command.

Once a command has been defined then

```
ICL> command . . .
```

is equivalent to typing

```
ICL> OBEYW taskname action . . .
```

with any parameters of command being appended to the OBEYW as the VALUE string sent to the task.

B.14 DEFPROC

DEFPROC command file [procedure]

Define a command which runs a procedure from a source file.

command - The command to be defined. An abbreviation may be specified in the form COM(MAND) where COM is the minimum acceptable abbreviation.

file - The source file containing the procedure.

procedure - (optional) The procedure to be executed. If omitted it defaults to command.

When *command* is issued, file.ICL is loaded and compiled (if it is not already loaded) and the procedure is called. Any parameters specified with the original command are passed to the procedure.

B.15 DEFSHARE

DEFSHARE command image [action]

Define a command which issues an OBEYW to an ADAM shareable image monolith

command - The command to be defined. An abbreviation may be specified in the form COM(MAND) where COM is the minimum acceptable abbreviation.

image - The logical name of the shareable image monolith which will execute the OBEYW.

action - (optional) The action to be executed. If omitted it defaults to command.

Once a command has been defined with DEFSHARE then

```
ICL> command . . .
```

is equivalent to typing

```
ICL> OBEYW taskname action . . .
```

with any parameters of command being appended to the OBEYW as the VALUE string sent to the task. The task must be linked into a shareable image monolith rather than a normal executable task.

B.16 DEFSTRING

DEFSTRING command equivalence_string

Associate a command with an equivalence string.

command - The command to be defined. An abbreviation may be specified in the form COM(MAND) where COM is the minimum acceptable abbreviation.

equivalence_string - The equivalence string for the command.

Issuing the command is equivalent to typing the equivalence string. Any parameters of the command are appended to the equivalence string.

B.17 DEFTASK

DEFTASK command [taskname]

Define a command which issues a SEND to an ADAM task

command - The command to be defined. An abbreviation may be specified in the form COM(MAND) where COM is the minimum acceptable abbreviation.

taskname - (optional) The task which will receive the SEND. If omitted it defaults to command.

Once a command has been defined using DEFTASK then

```
ICL> command . . .
```

is equivalent to typing

```
ICL> SEND action . . .
```

the parameters of command must supply the CONTEXT (GET, SET, OBEY, CANCEL) and the parameter or action name to be sent to the task.

B.18 DEFUSER

DEFUSER command image [routine]

Define a command which calls a user written subroutine in a shareable image.

command - The command to be defined. An abbreviation may be specified in the form COM(MAND) where COM is the minimum acceptable abbreviation.

image - The name of the shareable image containing the routine. This must be a logical name if the image is not in SYS$SHARE.

routine - (optional) The name of the subroutine to be called. This must be a universal symbol of the shareable image. If omitted it defaults to command.

The subroutine should have a single character string parameter in which it will receive the ICL command line parameter string. The string is received as typed, except for the substitution of bracketed expressions.

Typing *command* causes the shareable image to be activated dynamically (if it is not already loaded), and the subroutine to be called. The subroutine can return an error message to ICL by signalling a VMS condition. This will result in the ICL exception USERERR. The exception text will contain the message associated with the VMS status returned.

B.19 DELETE

DELETE proc

Delete procedure proc from the procedure table.

B.20 DISMOUNT

DISMOUNT dev [NOUNLOAD]

Dismount a tape previously mounted using the ICL MOUNT command. The NOUNLOAD parameter if specified means that the tape will be dismounted without unloading.

DISMOUNT may be abbreviated to DISMOU

B.21 DUMPTASK

DUMPTASK taskname

This command causes the ADAM task of name *taskname* to output a stack dump on the terminal from which it was loaded. For this command to be effective, the task must include a call to DTASK_SETDUMP(STATUS).

B.22 EDIT

EDIT proc

Edit procedure 'proc' using the selected editor. If the name of the procedure is unchanged during the editing session the new version replaces the old one. If the name is changed a new procedure is created and the old version of 'proc' remains unchanged. By default the TPU editor is used. The editor may be changed to EDT or LSE using the SET EDITOR command.

To edit a *file* use the command:

DCL EDIT filename

B.23 ENDOBEY

ENDOBEY (path) (messid)

Wait for completion of an ADAM I-task action initiated by STARTOBEY

path - The path associated with the action.

messid - The message identifier associated with the action.

B.24 EXIT

EXIT

EXIT from ICL and return control to DCL. On exiting from ICL a copy of all the current procedures is saved in the file SAVE.ICL

B.25 GET

GET taskname parameter (variable)

Get a parameter of an ADAM I-task and put the value into an ICL variable.

taskname - The name of the I-task.

parameter - The name of the parameter to be got.

variable - The ICL variable into which the value will be put.

B.26 GETGLOBAL

GETGLOBAL parameter (variable)

Get the value of a ADAM global parameter

parameter - The name of the global parameter to be got.

variable - An ICL variable to receive the result.

B.27 GETNBS

GETNBS nbsname (variable)

Get the value of a noticeboard item

nbsname - The name of the noticeboard item to be got.

variable - An ICL variable to receive the result.

The noticeboard name must correspond to a primitive item with one of the standard HDS types (*e.g.* _INTEGER). A dot notation is used to specify structure components (*e.g.* NOTICEBOARD.ITEM1.ITEM2). If the primitive item is an array only the first component is accessed.

B.28 GETPAR

GETPAR command parameter (variable)

Get the value of a parameter from the task's parameter file. This will only be relevant for parameters which are not *internal*.

command - A command name associated with an ADAM task by means of a DEFINE command. The task will usually be an A-task or monolith.

parameter - The name of the parameter to be got.

variable - An ICL variable to receive the result.

B.29 HELP

HELP [topic...]

Provide on line documentation on some aspect of ICL, or on some other topic which has been made available using the DEFHELP command.

topic... - An optional list of keywords specifying the path within the help library to the required information. The first keyword may be the *name* specified in a DEFHELP command.

B.30 INPUT

INPUT [prompt] (string)

Input a string from the terminal.

prompt - (optional) A prompt string to be output on the terminal

string - A variable into which the input string will be read.

B.31 INPUTI

INPUTI prompt (i) (j) ....

Input integers from the terminal

prompt - A prompt string to be output on the terminal.

i,j etc - The variables into which integers will be read.

B.32 INPUTL

INPUTL prompt (l) (m) ....

Input logical values from the terminal

prompt - A prompt string to be output on the terminal.

l,m etc - The variables into which logical values will be read.

The logical values may be input as YES, NO, TRUE, FALSE or any abbreviation of these, in either upper or lower case.

B.33 INPUTR

INPUTR prompt (x) (y) ....

Input real numbers from the terminal

prompt - A prompt string to be output on the terminal.

x,y etc - The variables into which the numbers will be read.

B.34 KEY

KEY keyname equivalence_string

Define an equivalence string for a key

keyname - The name of the key

equivalence_string - The equivalence string for the key. A # character may be used to indicate a RETURN character terminating the string.

The KEY command may be used at any time but the definition is only effective when screen mode is in use (however, the first SET SCREEN call in a session clears all key definitions).

B.35 KEYTRAP

KEYTRAP keyname

Specify trapping of a key

keyname - The name of the key

The KEYTRAP command may be used at any time but the definition is only effective when screen mode is in use (however, the first SET SCREEN call in a session clears all key definitions). The INKEY function is used to test for the pressing of a trapped key.

B.36 KEYUSER

KEYUSER keyname image routine

Associate key with a user written subroutine in a shareable image

keyname - The name of the key

image - logical name of the shareable image

routine - name of the subroutine to be called. This must be a universal symbol of the shareable image.

The KEYUSER command may be used at any time but the definition is only effective when screen mode is in use (however, the first SET SCREEN call in a session clears all key definitions). KEYUSER causes the specified routine to be called at AST level immediately the key is pressed. If necessary the shareable image is activated dynamically

B.37 KILL

KILL taskname

Kill an ADAM task

taskname - The name of the task to be killed.

B.38 KILLDCL

KILLDCL

Kill the DCL subprocess

B.39 KILLW

KILLW taskname

Kill an ADAM task and wait for it to die.

taskname - The name of the task to be killed.

B.40 LIST

LIST proc

List procedure ‘proc’ on the terminal

B.41 LOAD

LOAD file

Accept commands from ‘file.ICL’ rather than from the terminal.

‘file’ may specify a procedure previously saved using the SAVE command.

B.42 LOADD

LOADD exename [taskname] [priority]

Load an ADAM task into a detached process and wait for loading to complete.

exename - The name of the executable image file to be used for the task.

taskname - (optional) The name of the task to be created. If omitted it defaults to the file name part of exename.

priority - (optional) The priority for the created task.

B.43 LOADW

LOADW exename [taskname]

Load an ADAM task into a subprocess and wait for loading to complete:

exename - The name of the executable image file to be used for the task.

taskname - (optional) The name of the task to be created. If omitted it defaults to the file name part of exename.

B.44 MOUNT

MOUNT dev [(status)]

Mount a tape - equivalent to the DCL MOUNT/FOREIGN command. The tape is mounted as a foreign tape at its initialized density.

The optional parameter status is a variable in which the status of the mount operation will be returned. The success of the operation may be tested using the function OK(status).

MOUNT may be abbreviated to MOU.

B.45 NOREP

NOREP

Turn off reporting. Reporting is turned on by the REPORT command.

B.46 OBEYW

OBEYW taskname action . . .

Send an OBEY message to an ADAM task and wait for it to complete. Load the task as as a cached task if necessary.

taskname - The name of the task to which the obey message will be sent.

action - The action name to be obeyed.

. . . - Any parameters for the action.

B.47 OPEN

OPEN intname [filename]

Open an existing text file for input.

intname - The internal name by which the file will be known within ICL.

filename - The VMS file name of the file. If omitted the file name will be intname.DAT.

B.48 PRINT

PRINT p1 p2 ....

The parameters of PRINT are concatenated and printed on the terminal.

B.49 PROCS

PROCS

List the names of all current procedures.

B.50 PUTNBS

PUTNBS nbsname value

Write the value of a noticeboard item

nbsname - The name of the noticeboard item to be written.

value - The new value for the item.

The noticeboard name must correspond to a primitive item with one of the standard HDS types (*e.g.* _INTEGER). A dot notation is used to specify structure components (*e.g.* NOTICEBOARD.ITEM1.ITEM2). If the primitive item is an array, only the first component is accessed.

B.51 READ

READ intname (string)

Read a line from a text file.

intname - The internal name of the text file. The file must have been opened with the OPEN command.

string - A variable into which the input string will be read.

B.52 READI

READI intname (i) (j) ....

Read a line of integers from a text file.

intname - The internal name of the text file. The file must have been opened with the OPEN command.

i,j etc - The variables into which integers will be read.

B.53 READL

READL prompt (l) (m) ....

Read a line of logical values from a text file.

intname - The internal name of the text file. The file must have been opened with the OPEN command.

l,m etc - The variables into which logical values will be read.

The logical values may be input as YES, NO, TRUE, FALSE or any abbreviation of these, in either upper or lower case.

B.54 READR

READR intname (x) (y) ....

Read a line of real numbers from a text file

intname - The internal name of the text file. The file must have been opened with the OPEN command.

x,y etc - The variables into which the numbers will be read.

B.55 REPFILE

REPFILE [logfile] [DTNS]

Examine a logfile

logfile - Name of the logfile to be examined - defaults to ADAM_LOGFILE

DTNS - Four character string specifying which items in each log record are displayed. D = Date, T = Time, N = Name, S = String. Replace the item letter by anything else to stop the display of that item. DTNS is case independent.

B.56 REPORT

REPORT [logfile]

Turn on reporting. After reporting has been turned on, a log of input/output and ADAM message system traffic is written to the file specified by *logfile*. If the parameter is omitted the logical name ADAM_LOGFILE is used. If the file already exists, the log is appended to it.

The logfile may be examined using the REPFILE command or by the ADAM task, LISTLOG. LISTLOG is found along the ADAM_EXE search path and provides a means of obtaining readable hardcopy of the log. A DCL symbol LISTLOG is defined by ADAMSTART to run this task from DCL.

The command NOREP turns off reporting.

B.57 SAVE

SAVE proc

Save procedure 'proc' in the file 'proc.ICL'.

SAVE ALL

Save all procedures in file 'SAVE.ICL'.

Procedures save in this way may be reloaded using LOAD.

B.58 SAVEINPUT

SAVEINPUT lines filename

Save previous input lines in a text file. 'lines' specifies the number of lines to be saved, and 'filename' is the file in which they are saved. If 'filename' is omitted it defaults to SAVEINPUT.ICL in the default directory. If 'lines' is omitted the entire input buffer is saved.

B.59 SEND

SEND taskname context . . .

Send a GET, SET, OBEY or CANCEL message to an ADAM task. Load the task as as a cached task if necessary.

taskname - The name of the task to which the message will be sent. It must correspond with taskname in the ALOAD, LOADW or DEFINE command and include any directory specification given there.

context - The context (GET, SET, OBEY or CANCEL).

. . . - The remainder of the message. This will depend upon context.

SET parameter value

parameter - The name of the parameter whose value is to be set.
value - The value to be set. Type conversion will be done where necessary.

GET parameter

parameter - The name of the parameter. Its value will be displayed.

OBEY action parameters

action - The name of the action to be obeyed.
parameters - the parameter string for action.

CANCEL action parameters

action - The name of the action to be cancelled.
parameters - the parameter string for action.

OBEY, SET and GET contexts may be used with A-task monoliths. In that case, *action* is replaced by *atask* and *parameter* by *atask:parameter* where *atask* is the name of the individual A-task within the monolith.

B.60 SET

The SET command is used to control various features of the state of ICL.

B.60.1 SET ATTRIBUTES

SET ATTRIBUTES attributes

Sets the attributes for text written with the LOCATE command. *attributes* is a string containing any combination of the letters D (double size), B (bold), R (Reverse video), U (underlined) and F (flashing).

B.60.2 SET AUTOLOAD, SET NOAUTOLOAD

SET AUTOLOAD

SET NOAUTOLOAD

Switches on/off the automatic loading of tasks in response to an invocation. ICL's default is AUTOLOAD. If AUTOLOAD is not selected, tasks must be loaded explicitly with command ALOAD or LOADW.

B.60.3 SET CHECKPARS, SET NOCHECKPARS

SET CHECKPARS

SET NOCHECKPARS

Switches on/off the checking of parameters supplied to ICL procedures. ICL's default is CHECKPARS – in that case a procedure will not be executed unless the correct number of parameters is supplied. NOCHECKPARS allows the procedure to use the UNDEFINED function to test if the parameter was supplied and prompt if not. *Note that only trailing parameters may be omitted.*

B.60.4 SET EDITOR

SET EDITOR name

Sets the editor to be used for editing of ICL procedures. *name* must be one of TPU, EDT or LSE. (LSE may not be available on all systems). The default editor is TPU.

B.60.5 SET HELPFILE

SET HELPFILE library

Specifies a help library to be used by the ICL HELP system for topics other than those specified by a DEFHELP command. The library specification *library* should include a directory specification.

The default help library is ICLDIR:ICLHELP.

B.60.6 SET MESSAGES, SET NOMESSAGES

SET MESSAGES

SET NOMESSAGES

These commands control whether or not loading messages are output when ADAM tasks are loaded. By default loading messages are output.

B.60.7 SET PRECISION

SET PRECISION digits

Set the number of decimal digits precision for unformatted conversions of real values to strings. The default value is 6 and may be set to any value between 1 and 16.

B.60.8 SET PROMPT

SET PROMPT string

This command can be used to specify a prompt string to replace the default ICL> prompt.

B.60.9 SET SAVE, SET NOSAVE

SET SAVE

SET NOSAVE

This command controls whether ICL procedures are saved in a SAVE.ICL file when ICL exits. By default a SAVE.ICL file is created if there are any procedures to save.

B.60.10 SET SCREEN, SET NOSCREEN

SET SCREEN n

SET NOSCREEN

These commands select screen I/O mode or normal I/O mode. On the SET SCREEN mode the parameter n specifies the number of lines in the scrolling region. If n is omitted eight lines of scrolling region are allocated.

B.60.11 SET TRACE, SET NOTRACE

SET TRACE

SET NOTRACE

These commands turn on and off the tracing of procedure execution. By default tracing is off. When tracing is on, each line of the procedure is output on the terminal before execution.

B.61 SETGLOBAL

SETGLOBAL parameter value

Set the value of a ADAM global parameter

parameter - The name of the global parameter to be set.

value - The value for the parameter. It should not be enclosed in quotes unless the quotes are required as part of the global parameter value.

If the global parameter does not exist, one of type _CHAR*132 is created. This is usually OK but could give problems with _LOGICAL parameters. The CREATEGLOBAL command can be used to create global parameter storage of a specific type.

B.62 SETPAR

SETPAR command parameter value

Set the value of a parameter into the task's parameter file. This will only be relevant for parameters which are not *internal*.

command - A command name associated with an ADAM task by means of a DEFINE command. The task will usually be an A-task or monolith.

parameter - The name of the parameter to be set.

value - The value for the parameter.

If the parameter value is an HDS object name or device name it must be prefixed by @. SETPAR sets the current value of the parameter. Thus the VPATH or PPATH must include CURRENT for SETPAR to have any effect on subsequent execution of the command. Note that if the task is loaded, it will probably have its parameter file open and locked against ICL writing into it.

B.63 SIGNAL

SIGNAL name [text]

Signal an ICL exception

name - The exception name — any valid ICL identifier.

text - A message text associated with the exception.

Following SIGNAL an exception handler will be executed if one exists for the exception. Otherwise a message will be output, and control will return to direct mode.

B.64 SPAWN

SPAWN [dcl_command]

SPAWN with parameters concatenates the parameters to form a DCL command. creates a subprocess, and executes the DCL command in the subprocess.

SPAWN with no parameters creates a subprocess in which a series of DCL commands may be executed. Use LOGOUT to return to ICL.

In most cases DCL is a faster alternative to SPAWN as the subprocess does not have to be created for each new command. However SPAWN will do a few things which are not possible under DCL. SPAWNed commands will prompt for their parameters, for example, whereas commands issued using DCL will not.

B.65 STARTOBEY

STARTOBEY (path) (messid) task action value

Send an OBEY message to an ADAM task and return the path and message-id. Used in conjunction with ENDOBEY to set up multiple concurrent actions in a I-task.

path - An ICL variable to receive the path associated with the action.

messid - An ICL variable to receive the message identifier associated with the action.

task - The task to which the OBEY message will be sent.

action - The action to be executed.

value - Any parameters associated with the action.

B.66 TASKS

TASKS

List all loaded ADAM tasks.

B.67 VARS

VARS [proc]

Lists the variables of procedure 'proc' with their current types and values. VARS with no parameters lists the variables at direct mode.

B.68 WAIT

WAIT interval

Wait for a specified *interval* expressed in seconds.

B.69 WRITE

WRITE intname p2 p3 ....

The parameters of WRITE are concatenated and written to the text file of name *intname*. This file must have been previously opened with an APPEND or CREATE command.

Appendix C
ICL Exceptions

name	description
ADAMERR	An Error has occurred in an ADAM task. An associated message will give further information.
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 editor on a system on which LSE is not available, or an attempt to use TPU when the TPU shareable image is not accessible.
EOF	End of file encountered on text file operation.
FIGERR	Error in a FIGARO program, or an attempt to use a FIGARO command when the FIGARO shareable image is not accessible.
FLTDIV	Floating point division by zero.
FLTOVF	Floating point overflow.

name	description
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 — Please report the circumstances.

name	description
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.

Appendix D
ICL Syntax

The following gives a formal definition of the syntax of ICL. Note that there are in effect two levels to the ICL Syntax. One level describes how statements (strictly simple_statements as defined below) are built up, while a second level describes how statements (normally one per line) are combined to form control structures and procedures. The definitions of *if_block*, *loop_block* and *procedure* below form the second level syntax, and in these cases items on different lines in the definition, must appear on separate lines. The notation is as follows.

- Lower case words denote non terminal symbols of the grammar. *i.e.* symbols which are defined in terms of other symbols.
- Upper case words or other characters are terminal symbols. *i.e.* the basic lexical elements out of which the language is built.
- The construction `a | b` denotes a choice between two options.
- The construction `[ a ]` implies that a is optional.
- The construction `{ a }` implies that the symbol a may be repeated zero or more times.

Both spaces and layout are significant. Spaces may not appear within identifiers, or numbers. Spaces may be used as separators in parameter lists. layout is currently restricted to one statement per line. Thus the end of line character is effectively a statement separator.

```
letter  =   any of the letters A to Z or a to z

digit   =   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

binary_digit  =  0 | 1

octal digit  =  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hex_digit  =  digit | A | B | C | D | E | F | a | b | c | d | e | f

comment  =  ; anything   |
            { anything

unquoted_string  =  any sequence of characters not including a space,
                    comma or left parenthesis

string_delimiter  =  ' | "

open_string  =   any sequence of characters not including a string delimiter

string  =   string_delimiter  open_string  string_delimiter
            { string_delimiter  open_string  string_delimiter  }

identifier  =  letter  {  letter | digit | _  }
```

```
     ( certain identifiers have a special meaning within the language
       and are not available for general use

       these are  AND, OR, NOT, LOOP, WHILE, FOR, IF, ELSE, END, PROC,
                  TRUE, FALSE, BREAK, ENDIF, ELSEIF, ENDPROC, ENDLOOP,
                  EXCEPTION, ENDEXCEPTION.

       In identifiers a letter in upper or lower case is considered
       to be the same  )

integer  =  digit  {  digit  }

binary_integer  =  %B binary_digit  {  binary_digit  }

octal_integer  =  %O octal_digit  {  octal_digit  }

hex_integer  =  %X  hex_digit  {  hex_digit  }

scale_factor  =  E  integer  |
                 E  +  integer  |
                 E  -  integer

real  =  integer  .  {  digit  }  |
         integer  scale_factor  |
         integer  .  {  digit  }  scale_factor

number  =  real | integer | binary_integer | octal_integer | hex_integer

multiplication_operator  =  * | /

addition_operator  =  + | -

relational_operator  =   = | < | > | >= | <= | <>

logical_operator  =  AND | OR

function_call =  identifier  (  [ expression { , expression } ] )

primary  =  identifier  |  number  |  string  | function_call
            TRUE  |  FALSE  |  (  expression  )

factor  =  primary  {  **  primary  }

term    =  factor  {  multiplication_operator  factor  }

simple_expression  =  [  addition operator ]  term
                        {  addition_operator  term  }

relation  =  simple_expression  |
             simple_expression  relational_operator  simple_expression  |
             simple_expression : simple_expression [ : simple_expression ]

expression  =  relation  |
               NOT  relation  |
               relation  {  logical_operator  relation  }  |
               relation  {  &  relation  }

parameter  =  unquoted_string  |  string  |  (  expression  )
```

```
simple_statement  =   =  expression  |
                      identifier  =  expression  |
                      comment  |
                      simple_statement  comment

command   =   identifier  [  parameter  { [,]  parameter  }  ]


statement  =  simple_statement  |  command  |  if_block  |
              loop_block  |  BREAK

if_block  =   IF  expression
                { statement }
              [ ELSE IF  expression
                { statement } ]
              [ ELSE
                { statement } ]
              END IF

loop_clause =  WHILE expression  |
               FOR identifier = expression TO expression [ STEP expression ]

loop_block =  LOOP [loop_clause]
              { statement }
              END LOOP

procedure  =  PROC identifier [  identifier  { [,]  identifier }  ]
              { statement }
              { EXCEPTION identifier
                { statement }
                END EXCEPTION }
              END PROC
```