SUN/115.3

Starlink Project
Starlink User Note 115.3

A J Chipperfield, B D Kelly, S L Wright

3 September 2002

ADAM
Interface Module Reference Manual
Programmer's Manual

Abstract

ADAM Interface Modules provide an interface between ADAM application programs and the rest of the system.
This document describes in detail the facilities available with ADAM Interface Modules and the rules for using them. It is intended as a reference manual and should shed light on some of the finer points of the ADAM parameter system. Readers requiring an introduction to Interface Modules should read SG/4.

Contents

1 Introduction

This document gives a description of Interface Modules as used by application programs (tasks) running under ADAM with the parameter system based on the SUBPAR library. It describes Interface Modules and some related aspects of the parameter system in fine detail and is intended as a reference document – users requiring an introduction to the parameter system and Interface Modules should read SG/4 first.

For the purpose of the examples it is assumed that a user interface similar to ICL, the Interactive Command Language, is being used. Note that the actual use of keywords, prompts, *etc.* is determined by the user interface, which may or may not support the various features.

Useful associated documents are:

SG/4 ADAM — The Starlink Software Environment

SUN/134 ADAM — How to Write Instrumentation Tasks

SUN/92 HDS — Hierarchical Data System

SG/5 ICL — The Interactive Command Language

SUN/104 MSG and ERR — Message and Error Reporting Systems

This document applies to Unix ADAM Version 2.0 (VMS ADAM Version 2.1) and later.

2 The Interface Module

2.1 Description

The *Interface Module* is a collection of information which acts as a go-between for the application program on one hand, and the rest of the system on the other. It acts as an interface (hence the name) between the program and system so that neither has to know any details about the working of the other.This means that either side of the interface could be completely changed, without having to change anything on the other side. Only the Interface Module itself may have to be changed.

The main functions of the Interface Module are to:

- Specify the program's parameters and define a search path for parameter values.
- Specify actions within instrumentation tasks.
- Specify the format of the command line for the program.
- Specify the messages and prompts which the user sees.

2.2 Different Forms of an Interface Module – Compilation

An Interface Module can exist in two forms. When an application program is being developed, the information making up the Interface Module is normally put into an ordinary text file, where it can be easily inspected and edited. The file should have the same name as the program, but with a file extension of **.ifl**.

When the system interface for a program has become relatively stable, the Interface Module can be saved in compiled form – this leads to a more efficient running of the system. The compiled form has the same name as the source file, but a file extension of **.ifc**.

Strictly speaking, the source file is the *Interface File* although that term is often used to mean either source or compiled forms of the Interface Module.

The program **compifl** is provided to compile an Interface File. Perform the usual startup procedure for ADAM program development then type:

```
compifl filename
```

in response to the command language prompt. `filename` may optionally include the `.ifl` extension. `filename.ifl` will be compiled into `filename.ifc`.

2.3 Interface Module Search Path

When an ADAM program is first loaded, it searches for the required Interface Module, reads it and uses the information therein to set up the common blocks used by the parameter system to control its behaviour. The program will accept either source or compiled form of the Interface Module.

By default, the program will look for its Interface Module in the directory where the executable file was found. It looks firstly for a compiled file, `name.ifc` (where *name* is the name of the program's executable file); if that is not found, it looks for a source file, `name.ifl`

This behaviour can be altered by defining a list of directories in which the program will search before using the default. The process is fully described in Appendix B.

2.4 Basic Structure of the Interface File

2.4.1 Single Program

An Interface File for a single program consists of a number of *Specification Fields* contained between INTERFACE and ENDINTERFACE statements. There are four possible types of Specification Field under ADAM[1], the *Parameter Specification*, the *Help Library Specification*, the *Message Specification* and the *Action Specification*. They are all optional (depending upon the application).

Specification Fields may consist of a number of subfields as shown in the example below. *Examples in this document will use '.' to represent possible lines not shown.*

[1]Excluding the unused Program Specification ref. Appendix F.1

```
INTERFACE programname
   HELPLIB     HELP_TOP_LEVEL_SPEC
        .
   PARAMETER   parametername_1
      parfieldname   fieldvalue(s)
            .
   ENDPARAMETER
   PARAMETER   parametername_2
            .
   ENDPARAMETER
         .
   ACTION      actionname
      actfieldname   fieldvalue(s)
            .
   ENDACTION
         .
   MESSAGE     messagename
      TEXT     'string'
   ENDMESSAGE
      .
ENDINTERFACE
```

`programname` is a character string giving the name of the Interface Module. It must be the same as the name of the program and unique in any given ADAM system.

2.4.2 Monoliths

For efficiency, a group of related programs can all be linked into a single executable file called a *monolith* (see SG/4). If the monolith is to be run from ICL, the individual Interface Files must be concatenated into a single file for the whole monolith and the monolithic Interface File must have the same filename as the executable file. The structure of a monolithic Interface File is as follows:

```
MONOLITH monname
   INTERFACE programname_1
      .
      .
   ENDINTERFACE
   INTERFACE programname_2
      .
      .
   ENDINTERFACE
ENDMONOLITH
```

where `programname_1`, `programname_2`, *etc.* are the names of the individual programs. They must comply with the rules for action names (see Appendix A.1).

When monoliths are run directly from a Unix shell, the individual program Interface Modules are used.

2.4.3 Syntax

Details of Interface File parsing are given in Appendix A, but the following points should be sufficient for now.

- The file is interpreted as a sequence of *tokens*, usually separated by comma, space or newline.
- Tokens may be:
  - reserved words
  - constants (number, character, logical or !)
  - names (*i.e.* everything else).
- A character constant is a string of characters enclosed in single quotes. Where a token may be a character constant, a name (*i.e.* a single-token, unquoted string) or even a reserved word is usually also allowed. The term *character string* is used in this document where this situation pertains. An exception to this is where the token represents a parameter value (a default value, for example) – then character constants must be enclosed in quotes so that names can be distinguished from strings.
- Case is generally not significant.
- Extra white space is ignored (apart from within quoted strings) and it is recommended that blank lines and indentation are used for clarity.
- Comments may be introduced by # (apart from within quoted strings).
- STARLSE (see SUN/105) contains VAX Language Sensitive Editor templates for Interface Files.

3 Parameter Storage and the CURRENT Value

Each ADAM application program has zero or more *program parameters*. A program parameter is a value which the program gets from the 'outside world', or which the program wishes to make available outside itself. The value may be a primitive value (scalar or array) or a file or device name for example.

Before describing the way parameters are specified in the Interface File, it will be useful to say a little about the way in which parameter values are stored.

If a primitive value is supplied for parameters of a standard HDS type (see Section 4.3), storage of the defined type is created and, if possible, the value is converted to the storage type and stored. For other types, storage of the type of the supplied value is created.

If a name is supplied as a value for a parameter, the name is stored. For parameters of a standard HDS type, it is assumed to be the name of an HDS object of a suitable type and the storage type of the parameter is effectively the type of the named object.

The stored value of a parameter is known as the *current value*. It will normally[2] be in an HDS file known as the *parameter file*. A component of the right type, size and shape is created to hold the given parameter value – there is a special structure to hold values which are names. Each program or monolith has its own parameter file which is retained between sessions. The

[2]The exception is 'internal' parameters (see Section 4.10) which are stored in memory.

parameter file name is `name.sdf` (where `name` is the name of the executable file) and, by default, it will be found in subdirectory `adam` of the user's login directory.

Note that the current value is usually thought of as the last-used value, however:

- The given value may have been rejected by the parameter system or program after having been stored. This will be the case if it fails a range check, for example.
- The current value may have been written into the parameter file by external means, the ICL SETPAR command, for example.

4 The Parameter Specification

4.1 Introduction

Each program parameter must have a *Parameter Specification* in the Interface File. The Parameter Specification is of the form:

```
PARAMETER parametername
   parfieldname fieldvalue(s)
         .
ENDPARAMETER
```

where `parametername` is a character string giving the program parameter name which is used in the program (in a call to DAT_ASSOC or PAR_GETnx *etc.*). Case is not significant in parameter names. For a discussion of valid parameter names, see Appendix A.1.

The possible subfields in the Parameter Specification are described in detail below – they are all optional and, apart from TYPE, which must appear before DEFAULT, RANGE or IN fields, the order is unimportant.

4.2 Summary of Parameter Specification Subfields

The Parameter Specification subfields are described in detail in the following sections. This is a classified list to show the features available.

Data Type and Access Fields

Field	Specified Attribute	Default
TYPE	Type of the parameter	univ
PTYPE	Parameter is a Device Parameter	Normal Parameter
ACCESS	Access required to the parameter	update

Command Line Control Fields

Field	Specified Attribute	Default
POSITION	Command line parameter position allocated	
KEYWORD	User's name for the parameter	parameter name

Value Control Fields

Field	Specified Attribute	Default
DEFAULT	Static default value	
ASSOCIATION	Associated global parameter	
VPATH	Search path for parameter value	prompt
RANGE	Range of values the parameter may take	
IN	Set of values the parameter may take	

Prompt Control Fields

Field	Specified Attribute	Default
PROMPT	String to be used as parameter prompt	parameter name
PPATH	Search path for prompt default value	'dynamic,default'
HELP	Single-line parameter help message	
HELPKEY	Source of multi-line parameter help information	

4.3 The TYPE Field

This field specifies the type of the parameter. This field is used by the system to determine the type of storage to be used for the parameter and its default values, *etc*. If a primitive value of a different type is supplied, it will be converted to the specified type for storage if possible. If conversion is not possible, an error is reported.

It is important to distinguish between the type of the parameter and the type of the value to be obtained from it by the program.

The field is of the form:

```
TYPE datatype
```

where `datatype` is a character string giving the required type of the parameter. `datatype` may be:

- A 'standard' HDS primitive type (*i.e.* _INTEGER, _REAL, _DOUBLE, _LOGICAL, _CHAR).

- LITERAL – formerly used to specify that the parameter is of type _CHAR and to force the parameter system to accept unquoted character strings as character values rather than HDS object names. This is now the standard behaviour for parameters of type _CHAR so datatype LITERAL is no longer required.
- A 'user-defined' type (IMAGE, SPECTRUM *etc.*).
- UNIV – used by convention when the parameter type is unimportant. Actually this is treated the same as other 'user-defined' types. Parameter storage corresponding with the type of the value given will be created.
- A 'facility-recommended' type (TAPE for MAG, FILENAME for FIO *etc.*). These are used by convention – actually any user-defined type will do.

If the TYPE field is omitted, type UNIV is assumed.

Notes:

(1) Internally only the 'standard' primitive types (_LOGICAL, _INTEGER, _REAL, _CHAR, _DOUBLE) are differentiated – all other types are treated the same.

(2) Only values of the standard types, or names, can be given in the Interface File or as parameter values at run time.

(3) The dimensionality of parameters is not declared in the Interface File, but is defined by the value given at run time. The given dimensionality must be acceptable to the program (*e.g.* when it does a call to PAR_GETnx).

(4) The PAR_GETnx/_PUTnx routines only GET/PUT values of the standard HDS types – other types have to be handled by calls to the appropriate facility_ASSOC routine (DAT_ASSOC for example). Where primitive values are being handled, conversion between the value type and the storage type will be performed if necessary and possible.

Example

Suppose there is a program DISP that draws an image on an image display device. It has three parameters:

IMAGE the dataset containing the image to be drawn,

DISPLAY the name of the image display device, and

LIMITS an integer array of limiting values.

The Interface File might be:

```
INTERFACE DISP
   .
   PARAMETER IMAGE
         .
      TYPE IMAGE
         .
```

```
      ENDPARAMETER
      PARAMETER DISPLAY
            .
         TYPE IMAGEDISPLAY
            .
      ENDPARAMETER
      PARAMETER LIMITS
            .
         TYPE _INTEGER
            .
      ENDPARAMETER
         .
   ENDINTERFACE
```

If the user specified a value of `[1.2,511.8,255.6,512.1]` for the LIMITS array, *i.e.* a _REAL array where type _INTEGER is specified for the parameter, the system knows that this is a valid conversion and can perform it for the user (*i.e.* `[1,511,255,512]` is stored).

Similarly, the program could try to obtain a _REAL array from the parameter storage (by calling PAR_GET1R). In that case, the array would be converted back to _REAL although the numbers supplied by the user would have been truncated (*i.e.* `[1.0,511.0,255.0,512.0]` would be supplied to the program).

4.4 The PTYPE Field

Most parameters are associated with values of one type or another, *e.g.* primitive data types, or data-system objects. These are known as *normal parameters*.

However, there is a class of parameters which are intimately associated with some type of device (*e.g.* a magnetic tape drive, or a graphics terminal). These are known as *device parameters*. Device parameters are typically used in calls such as GKS_ASSOC. A parameter is defined as a device parameter by the PTYPE field.

```
   PTYPE  parameter-type
```

where `parameter-type` is a character string which can currently only be DEVICE.

The precise type of the device can be indicated by the TYPE field. The specification for parameter DISPLAY in our example above would then be:

```
   PARAMETER DISPLAY
         .
      PTYPE DEVICE
      TYPE GRAPHICS
         .
   ENDPARAMETER
```

The value of a device parameter is a name which can be translated to define the device to be used. For details of possible values, see the appropriate facility description, SGS, GKS, FIO *etc.*

Currently PTYPE is not used by the system – only the TYPE is of significance and there anything other than a standard HDS type will have the same effect.

4.5 The ACCESS Field

This field specifies the mode of access which the program requires to the parameter. It is used to restrict access to parameters handled by PAR routines or by DAT_ASSOC.

Note that the field has no effect for facilities such as NDF, FIO and GKS which only use the parameter system to get the name of a file or device which is accessed by other means. The access mode applies to the parameter itself, not to any file or device, *etc.* that the parameter refers to.

The field has the form:

```
ACCESS mode
```

where `mode` is a character string which may be:

READ which gives read access.

WRITE which gives write access.

UPDATE which gives read and write access.

Thus if a program does not modify the parameter, READ should be specified in the Interface File. If the value is to be written or modified, then WRITE or UPDATE should be specified.

If the ACCESS field is omitted, UPDATE is assumed.

Example

The access fields for the parameters in our DISP example could be:

```
INTERFACE DISP
   .
   PARAMETER IMAGE
         .
      TYPE IMAGE
      ACCESS READ
         .
   ENDPARAMETER
   PARAMETER DISPLAY
         .
      PTYPE DEVICE
      TYPE IMAGEDISPLAY
         .
   ENDPARAMETER
   PARAMETER LIMITS
         .
      TYPE _INTEGER
      ACCESS READ
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

4.6 The POSITION Field

Defines the *parameter position* on the command line at which values for this parameter may be given.

There are a limited number (See Appendix E) of command line positions available to be allocated. Not every parameter need be allocated a position, but the positions allocated must form a contiguous set starting at 1. Each position may have only one parameter allocated to it.

Note that, for ADAM I-tasks, the POSITION field may be overridden by an individual action NEEDS list (see the ACTION specification, Section 7).

The field is of the form:

```
POSITION n
```

where `n` is an integer constant which specifies the allocated position.

If the POSITION field is omitted, no command line position is allocated to this parameter.

Example

Again taking our DISP example, if the positions were specified as follows:

```
INTERFACE DISP
        .
   PARAMETER IMAGE
        .
      POSITION 1
        .
   ENDPARAMETER
   PARAMETER DISPLAY
        .
      POSITION 2
        .
   ENDPARAMETER
   PARAMETER LIMITS
        .
      POSITION 3
        .
   ENDPARAMETER
      .
ENDINTERFACE
```

and the user gives the command:

```
disp iue.image.swp2099 ikon [1,511,255,512]
```

Then the HDS object `iue.image.swp2099` would be associated with parameter IMAGE, XWINDOWS would be used as the DISPLAY and LIMITS would take the value `[1,511,255,512]`.

4.7 The KEYWORD Field

This field defines the name by which the parameter is known to the person running the program and which can be used on the command line to specify a value for the parameter. It will also be used to refer to the parameter in prompts and error messages. Case is not significant in keywords – they are always displayed in upper case.

The field is of the form:

```
KEYWORD  name
```

where `name` is a valid keyword[3].

This completes the separation between the program's view of parameters, and the user's view of them. It is possible for the programmer to re-write the program using completely different parameter names, but the user's view could be kept the same by just changing the parameter names in the Interface File. Similarly, the user's view of the program can be changed by just changing the KEYWORD fields.

If the KEYWORD field is not specified for a parameter, then the system will use the parameter name as the keyword.

Example

If the names of the parameters in the DISP example were changed, the user's view could be preserved by specifying keywords as follows (*Note: the example is not intended as an illustration of common practice – it is only to illustrate the principle*):

```
INTERFACE DISP
   .
   PARAMETER P1
         .
      POSITION 1
      KEYWORD IMAGE
         .
   ENDPARAMETER
   PARAMETER P2
         .
      POSITION 2
      KEYWORD DISPLAY
         .
   ENDPARAMETER
   PARAMETER P3
         .
      POSITION 3
      KEYWORD LIMITS
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

[3]See Appendix A.1 for a discussion of valid keyword names.

The user could then specify:

```
disp display=ikon image=iue.image.swp2099 limits=[1,511,255,512]
```

Note that keyword-style parameter specifiers are ignored in calculating *position*. Therefore, with the above Interface File, the same effect would be produced by the command:

```
disp display=ikon iue.image.swp2099 limits=[1,511,255,512]
```

Logical Keywords

If a parameter is of type _LOGICAL, its keyword alone may be used on the command line to specify the value TRUE for the parameter and the keyword prefixed by NO to specify the value FALSE.

For example, assume a program TEST with a logical parameter, keyword SWITCH, then:

`test switch` is equivalent to `test switch=TRUE`

and

`test noswitch` is equivalent to `test switch=FALSE`

4.8 The DEFAULT Field

This field is used to specify a *static default* value for a parameter. The static default value is one of the options available to be used as the parameter value.

The field is of the form:

```
DEFAULT default_value
```

where `default_value` may be:

- A constant (number, character or logical).
- A name.
- An array. (Currently only 1-D arrays may be given, and the run-time syntax for arrays, involving square brackets, cannot be used – they must be specified as a sequence of constants and may be enclosed in parentheses.)
- The null value (!).

The field is terminated by the next reserved-word token, therefore names which correspond with a reserved word cannot be given unless enclosed in quotes.

Note that DEFAULT cannot be specified before the TYPE field has been declared and the value specified must be appropriate for the specified type. A character constant (quoted string) may be given as `default_value` for a non-primitive type – the string within the quotes will be used as the 'name' when required. This can be useful if non-standard strings are to be given as defaults (*e.g.* lists for the GRP facility handling parameters declared as TYPE NDF).

Example

```
INTERFACE DISP
      .
   PARAMETER IMAGE
         .
      TYPE IMAGE
      DEFAULT IUE.IMAGE.SWPTEST
         .
   ENDPARAMETER
   PARAMETER DISPLAY
         .
      PTYPE DEVICE
      TYPE IMAGEDISPLAY
      DEFAULT XWINDOWS
         .
   ENDPARAMETER
   PARAMETER LIMITS
         .
      TYPE _INTEGER
      DEFAULT (1,512,1,512)
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

4.9 The ASSOCIATION Field

This field is used to specify that another parameter is to be associated with this parameter. The value of the associated parameter is one of the options available to be used as the value of this parameter. Furthermore, the value of a parameter can be used to automatically update the value of its associated parameter on successful completion of the program.

This behaviour is primarily of relevance to ADAM A-tasks. I-tasks can obtain parameter values from an association, but do not write parameters on completion.

The field is of the form:

```
ASSOCIATION  association-specification
```

where `association-specification` is a character string consisting of two parts – an association operator followed by a parameter specifier.

There are three association operators:

- <- The associated parameter may be used by the program (see Sections 4.10 and 4.14) but will not be updated by it.
- -> The associated parameter will be set to the value of the program parameter if:
  - The program has ended successfully, *i.e.* STATUS = SAI__OK.
  - The program parameter has a value and has not been cancelled.

  - The program is an A-task or A-task monolith.

- <-> combines <- and ->.

Currently, only *global* parameters may be associated with program parameters – they are stored in the GLOBAL data structure and provide a common pool of parameter values for programs in an integrated system. Global parameters can be created by the association write mechanism or by an ICL CREATEGLOBAL or SETGLOBAL command. They may also be read into ICL variables by the GETGLOBAL command. It is necessary to be careful when multi-tasking in ADAM because an error will occur if two programs attempt to update the GLOBAL data structure simultaneously.

The specifier of a global parameter is of the form:

```
GLOBAL.parameter_name
```

If the ASSOCIATION field is omitted, no association is made.

Example

```
INTERFACE DISP
   .
   PARAMETER IMAGE
         .
      ASSOCIATION <->GLOBAL.IMAGE
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

This gives the parameter DISP.IMAGE read and write access to the global parameter GLOBAL.-IMAGE.

4.10 The VPATH Field

This field is used to specify the way in which a value is obtained for a parameter if the user does not specify a value (on the command line, or by an ICL SEND SET command, for example) before it is required by the program. In that case, there are several potential sources for parameter values and different applications will want to use different sources for their various parameters.

The VPATH field is the means whereby this order of searching for parameter values is specified.

The field is of the form:

```
VPATH  value-resolution-path
```

where `value-resolution-path` is a character string which gives a path used for searching for a parameter value. It consists of a set of *path specifiers*. If there are two or more path specifiers, they must be separated by commas and the string must be enclosed in single quotes.

The valid specifiers are:

CURRENT Use the current (last-used) value of the parameter. (See Section 3 for more information on the current value.)

DYNAMIC Use the dynamic default value specified by the program. Dynamic defaults are set by the program calling a subroutine such as PAR_DEFnx or DAT_DEF specifying the required values.

DEFAULT Use the static default specified in the Interface Module (see Section 4.8).

GLOBAL Use the value of the associated parameter (see Section 4.9, The ASSOCIATION Field).

PROMPT Prompt the user and obtain a value. (See also Section 4.14, The PPATH Field.) There is an implied PROMPT at the end of every VPATH. *Note that PROMPT will always give a result to the program, even if it is a bad status, therefore PROMPT only makes sense as the last thing on the path.*

NOPROMPT Give up trying to get a value and return status PAR__NULL. This only makes sense as the last specifier in the path and will prevent a prompt being issued as a last resort if the value-resolution-path is exhausted.

INTERNAL This specifier can only be used as the first and only specifier on the VPATH. A `value-resolution-path` of 'DYNAMIC,CURRENT,NOPROMPT' is implied[4]. 'Internal' storage will be used to hold the parameter values. Scalar values are stored in memory to give enhanced performance; array values are stored in the parameter file.

If the value for a parameter has not been specified when it is required, the system looks at the VPATH specification, picks out the first path specifier and tries to find a value from this source. If a value is not found, the next path specifier is extracted and another search is made. This process continues until a value is found, the specifier is NOPROMPT or the path specification runs out. If the path specification is exhausted, the user is prompted for a value.

Note that the VPATH field is only used the first time a parameter value is obtained. If the parameter is cancelled before another attempt to 'get' a value, a prompt will be issued, even if NOPROMPT is put on the VPATH. If the parameter is not cancelled, the existing value will be returned again.

If the VPATH field is omitted, VPATH PROMPT is assumed.

Example

Suppose the parameters of DISP are specified as follows:

```
INTERFACE DISP
   .
   PARAMETER IMAGE
         .
      VPATH GLOBAL
      ASSOCIATION <->GLOBAL.IMAGE
         .
   ENDPARAMETER
   PARAMETER DISPLAY
```

[4]Old-style D-task parameters should have VPATH 'INTERNAL' specified as D-tasks cannot prompt for values.

```
        .
      VPATH 'CURRENT,DEFAULT'
      DEFAULT XWINDOWS
        .
    ENDPARAMETER
    PARAMETER LIMITS
        .
      VPATH DYNAMIC
        .
    ENDPARAMETER
      .
ENDINTERFACE
```

Then, if the user just types:

```
disp
```

(*i.e.* does not give any parameter values), the following actions occur:

When the program asks for the value of the IMAGE parameter, no value has been given so the first part of the VPATH specification is extracted (GLOBAL). This path specifier tells the system to look for a value of the associated parameter (GLOBAL.IMAGE). If it has a value, this value is taken as the value of IMAGE; if not, the resolution path is exhausted so the user will be prompted for a value for IMAGE.

When the program asks for a value of DISPLAY, the specifier CURRENT is extracted from the VPATH. If DISPLAY has a current value, the same value is used again; if not, the static default value (XWINDOWS) specified in the Interface File is taken.

Similarly, when the program asks for the value of LIMITS, the system attempts to use the dynamic default suggested by the program. If no dynamic default has been set, the user is prompted for a value for LIMITS.

Run-time Modification of VPATH Action

The user can modify the action of VPATH at run time by using the special command-line keywords PROMPT and/or RESET.

PROMPT causes the VPATH to be ignored and a prompt to be issued for any required parameters which have not already been given values.

RESET makes the system ignore CURRENT on the VPATH. See also Section 4.14 for the effect of RESET on prompts.

For example, assuming the Interface File above,

```
disp prompt
```

causes all parameters to be prompted for, irrespective of VPATH.

```
disp reset
```

causes the system to ignore 'CURRENT' in the VPATH for parameter DISPLAY, and use the DEFAULT value.

4.11 The RANGE Field

This field is used to specify a range of permitted values for the parameter. The RANGE values will also be used if MIN or MAX is specified as the parameter value in the absence of any dynamic minimum or maximum values set by the program.

RANGE can only be specified for parameters of standard primitive type other than _LOGICAL, and is only relevant if the parameter value is a scalar. A RANGE field will also constrain any dynamic minimum or maximum values which the program may set.

A parameter cannot have both a RANGE and an IN field and RANGE cannot be specified before TYPE has been declared.

Range checking is carried out when a program attempts to get the value of a parameter or set a minimum or maximum value. No range checking occurs when putting a value. For _CHAR parameters, all values are converted to upper case for checking and the ASCII collating sequence is used.

If the constraints are violated, the system reports the error and, if the violation was on getting a parameter value, prompts for another value unless the parameter had VPATH INTERNAL, in which case status SUBPAR__OUTRANGE is returned.

The field is of the form:

```
RANGE min, max
```

where `min` and `max` are two character or number constants specifying the minimum and maximum values for the parameter – the values must be convertible to the type of the parameter.

If `min > max`, the value must not lie between them. The values themselves are always acceptable.

Example

```
PARAMETER EXPONENT
      .
   TYPE _REAL
   RANGE 0.001, 1000.0
      .
ENDPARAMETER
```

4.12 The IN Field

This field is used to specify a set of values which the parameter may take. IN can only be specified for parameters of standard primitive type other than _LOGICAL, and is only relevant if the parameter value is a scalar.

A parameter cannot have both a RANGE and an IN field and IN cannot be specified before TYPE has been declared.

Checking is carried out when a program attempts to get the value of a parameter.

No checking occurs when putting a value. For _CHAR parameters the check is case independent.

If the constraints are violated, the system reports the error and prompts for another value unless the parameter had VPATH INTERNAL, in which case status SUBPAR__OUTRANGE is returned.

The field is of the form:

```
IN set-of-values
```

where `set-of-values` consists of a list of character or number constants specifying the valid values for the parameter. The specified values must all be convertible to the type of the parameter. The set of values is terminated by the next reserved-word token, therefore any required string which corresponds with a reserved word must be enclosed in quotes.

Example

```
PARAMETER FILTER
      .
   TYPE '_CHAR'
   IN 'R', 'I', 'J'
      .
ENDPARAMETER
```

4.13 The PROMPT Field

This field is used to specify a prompt string for the parameter[5].

The field is of the form:

```
PROMPT  text
```

where text is a character string (*n.b. it must be enclosed in quotes if it consists of more than one token*).

If the PROMPT field is omitted, the parameter keyword is used as the prompt.

Example

```
INTERFACE DISP
   .
   PARAMETER IMAGE
         .
      VPATH PROMPT
      PROMPT 'Image to be displayed'
         .
   ENDPARAMETER
   PARAMETER DISPLAY
         .
      VPATH PROMPT
      PROMPT 'Image display device'
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

[5]Use of the prompt string is actually a function of the user interface. The effect with ICL, SMSICL, DCL and Unix shells is described here.

Suppose that the user has given the command:

```
disp image=iue.image.swp2099
```

The program will take the specified value as the value of the IMAGE parameter. When it requires the DISPLAY parameter value, none has been specified on the command line so, as the VPATH field specifies that the user is to be prompted for a value, the system generates a prompt line such as:

```
DISPLAY - Image display device >
```

to which the user has to reply with a suitable value. (The prompt may also have a suggested value for the parameter, obtained via the PPATH field, see below).

4.14 The PPATH Field

This field specifies a path used for searching for a *suggested value* [6]. The suggested value will be displayed as part of the prompt string and used as the parameter value if the user responds to the prompt by hitting the carriage return key. Hitting the TAB key will make the suggested value available in the terminal input buffer where it can be edited with the normal line editing commands.

The field is of the form:

```
PPATH  value-resolution-path
```

where `value-resolution-path` is a character string consisting of a set of *path specifiers*. If there are two or more path specifiers, they must be separated by commas and the string must be enclosed in single quotes.

The valid specifiers are:

CURRENT Use the current value of the parameter. (See Section 3 for more information on the current value.)

DYNAMIC Use the dynamic default value specified by the program. Dynamic defaults are set by the program calling a subroutine such as PAR_DEFnx or DAT_DEF specifying the required values.

DEFAULT Use the static default value specified in the Interface Module (see Section 4.8).

GLOBAL Use the value of the associated global parameter (see Section 4.9, The ASSOCIATION Field).

If the suggested value for a parameter is required, the system looks at the PPATH specification, picks out the first path specifier and tries to find a value from this source. If a value is not found, the next path specifier is extracted and another search is made. This process continues until a value is found, or until the path specification runs out. The value is then passed, together with the prompt string, in a parameter request message to the user interface.

If the PPATH field is omitted, or fails to give a value, `'DYNAMIC,DEFAULT'` is used. If this fails to give a value, there is no suggested value.

[6] Use of the suggested value is actually a function of the user interface. The effect with ICL, SMSICL, DCL and Unix shells is described here.

Example

```
INTERFACE DISP
   .
   PARAMETER IMAGE
         .
      PROMPT 'Image to be displayed'
      PPATH 'CURRENT,DEFAULT'
      DEFAULT IUE.IMAGE.SWPTEST
         .
   ENDPARAMETER
   PARAMETER DISPLAY
         .
      PROMPT 'Image display device'
      PPATH GLOBAL
      ASSOCIATION <-GLOBAL.DEVICE
         .
   ENDPARAMETER
   PARAMETER LIMITS
         .
      PROMPT 'Pixel limits'
      PPATH 'DYNAMIC,DEFAULT'
      DEFAULT 1,511,255,512
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

Assuming that:

(1) There is no current value for IMAGE.

(2) GLOBAL.DEVICE has the value XWINDOWS.

(3) The program has not specified a dynamic default for LIMITS.

Then, with the above Interface File, if the user just types:

```
disp
```

(*i.e.* does not give any parameter values), and the VPATH field results in the user being prompted, the three prompts would be something like:

```
IMAGE - Image to be displayed /@IUE.IMAGE.SWPTEST/ >
DISPLAY - Image display device /@XWINDOWS/ >
LIMITS - Pixel limits /[1,511,255,512]/ >
```

Note the '@' in the suggested values is inserted by the system to avoid ambiguity between names and character strings.

Run-time Modification of PPATH Action

The user can modify the operation of the PPATH at run-time by using the special command-line keywords RESET and/or ACCEPT.

RESET Causes CURRENT to be ignored on the PPATH.

ACCEPT (or \) causes the suggested value to be used for any required parameters which would otherwise be prompted for. The user may also respond to a prompt with \. The suggested value will be taken for the prompted parameter and for any remaining parameters. It is also possible to force ACCEPT for individual parameters by putting `keyword=ACCEPT` or `keyword=\` on the command line (where `keyword` is the keyword of the parameter).

If there is no suggested value, the prompt will be issued anyway.

E.g. with the Interface File above, the command:

```
disp reset accept
```

will cause the static default value, rather than the current value, to be used as the suggested value for IMAGE, and that value to be used as the parameter value without prompting.

4.15 The HELPKEY Field

This field is used to specify a help file and a module within it at which *multi-line* help text for the parameter may be found[7]. Text read from a help file is known as *multi-line* help although it could in fact be only one line.

The field is of the form:

```
helpkey help_specifier
```

where `help_specifier` is a character string of the form:

```
filename key1 key2 etc...
```

specifying that the text to be displayed may be found in the help file `filename` and the module defined by `key1 key2 etc...` (but see also the HELPLIB Specification, Section 5). For more information on multi-line help and a discussion of help filenames, see Appendix D.

`help_specifier` may also be `*` or `'*'`, in which case a default value of:

```
interface_name PARAMETERS parameter_name
```

will be used, where `interface_name` is the name specified in the current INTERFACE field. (Use of this default implies that there is an associated HELPLIB Specification.)

If the user is prompted for a parameter value, and responds with '??', the specified text is displayed at the terminal and the user will be left in the help system and prompted for further topics of interest.

If the user responds with '?' and there is no HELP field specified for the parameter the effect is the same except that the user will be re-prompted for the parameter value immediately (except from SMSICL).

[7] The precise effect of the HELPKEY field will depend upon the user interface in use. The effect with ICL, SMSICL, DCL and Unix shells is described here.

Example

The previous example is therefore better written:

```
INTERFACE DISP
   .
   PARAMETER IMAGE
         .
      HELP 'This is the name of an image dataset to be displayed'
         .
   ENDPARAMETER
      .
      .
   PARAMETER LIMITS
         .
      HELPKEY 'DISP_DIR:DISP DISP PARAMETERS LIMITS'
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

4.16 The HELP Field

This field is used to specify some help text for a parameter[8].

This field is of the form:

```
HELP help-specifier
```

where `help-specifier` is a character string giving help information about the parameter, it may be either a single line of text to be displayed (*single-line help*) or a string of the form:

```
%filename key1 key2 etc...
```

specifying that multi-line help may be found in the help file `filename` and the module defined by `key1 key2 etc...`

Note that the preferred way of specifying multi-line help is using the HELPKEY field and that single-line help is not generally considered very useful.

If the user is prompted for a parameter value, and responds with '?', the specified text is displayed at the terminal and the user is re-prompted. (Except in the case where SMSICL outputs multi-line help – then the user is left in the help system and prompted for further topics of interest. When the user exits from the help system, the parameter prompt will reappear.)

If the user responds with '??' and there is no HELPKEY field specified for this parameter, the effect is the same except that if multi-line help is output the user will be left in the help system and prompted for further topics.

[8] The precise effect of the HELP field will depend upon the user interface in use. The effect with ICL, SMSICL, DCL and Unix shells is described here.

Example

```
INTERFACE DISP
   .
   PARAMETER IMAGE
         .
      HELP 'This is the name of an image dataset to be displayed'
         .
   ENDPARAMETER
      .
      .
   PARAMETER LIMITS
         .
      HELP '%disp_dir:disp disp parameters limits'
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

5 The HELPLIB Specification

This specification may be used to specify a 'top level' for all subsequent HELPKEY fields until another HELPLIB specification is found. The specification may appear anywhere in an Interface File.

The field is of the form:

```
HELPLIB help_top_level_spec
```

where `help_top_level_spec` is a character string to be inserted (with an intervening space) in front of any string specified in subsequent HELPKEY fields as the Interface File is parsed – it may be a blank character constant to nullify the effect of an earlier HELPLIB field.

Thus, assuming that the help file for the program DISP is DISP_DIR:DISP and that help text for each parameter is held in subtopic *parameter-name* of subtopic PARAMETERS of topic DISP, then the Interface File for program DISP could be:

```
INTERFACE DISP
# Specify the top level of help
   HELPLIB 'DISP_DIR:DISP DISP PARAMETERS'
   PARAMETER IMAGE
         .
      HELPKEY 'IMAGE'
         .
   ENDPARAMETER

# Specify top level help for use with default helpkey
   HELPLIB DISP_DIR:DISP
   PARAMETER DISPLAY
         .
```

```
      HELPKEY *
         .
   ENDPARAMETER

# Just as an example, use the blank specifier
   HELPLIB ''
   PARAMETER LIMITS
         .
      HELPKEY 'DISP_DIR:DISP DISP PARAMETERS LIMITS'
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

Note that the use of multiple HELPLIB fields here is purely to show the effects – there would normally be only one HELPLIB specification in an Interface File.

6 The MESSAGE Specification

If a program uses the ADAM message or error systems (MSG or ERR, see SUN/104), then a message text is required. The Message Specification can be used to specify text to be used in preference to text supplied by the program in the call to the MSG or ERR routine. Each of the Message Parameter names used in the program can have a Message Specification in the Interface File. The Message Specification is of the form:

```
MESSAGE parametername
  TEXT  string
ENDMESSAGE
```

where `parametername` is a character string specifying the Message Parameter name which is used in the call to an ERR or MSG subroutine. The name must be different from any other Program or Message Parameter name specified for the program.

`sting` may by anything acceptable to the ERR or MSG routine (*i.e.* it may include tokens *etc.*).

If the Message Specification for a given Message Parameter is omitted, the text specified in the MSG or ERR routine will be used.

Example

```
MESSAGE APP_OUTMESS
   TEXT 'Calculated value is ^VALUE'
ENDMESSAGE
```

7 The ACTION Specification

The concept of ACTION is specific to instrumentation tasks (see SUN/134). The task is capable of obeying or cancelling a number of distinct commands known as actions. It does this, for example, in response to the ICL command:

```
SEND taskname context actionname parameter_list
```

where `context` is `OBEY` or `CANCEL`.

Actions have to be declared in the Interface File, following the parameter declarations. The Action Specification is of the form:

```
ACTION actionname
   fieldname fieldvalue
   context specifications
      .
ENDACTION
```

Where `actionname` is a character string specifying a valid action name. The same rules should be applied to action names as are applied to parameter names (see Appendix A.1).

7.1 The Context Specification and NEEDS Field

The Context Specifications allow command-line parameter positions to be allocated differently for different actions and contexts of the task. The Context Specification is of the form:

```
   context
      NEEDS parametername_1
      NEEDS parametername_2
       .
   ENDcontext
```

where `context` is `OBEY` or `CANCEL`.

The order of the NEEDS fields within the Context Specification defines the order in which parameters may be specified on the command line for this particular action and context.

If no NEEDS list is specified, positions will default to those specified in the Parameter Specification.

Example

```
ACTION FIRST
   OBEY
      NEEDS PAR1
      NEEDS PAR2
      NEEDS PAR3
   ENDOBEY
   CANCEL
      NEEDS PAR2
   ENDCANCEL
ENDACTION
```

For historical reasons, NEEDS specifications can have RANGE or IN constraints but these have no effect.

Example

```
ACTION GET_FILTER
   OBEY
      NEEDS FILTER IN 'R', 'I', 'J'
      NEEDS DEAD_TIME RANGE 1.0, 5.5
   ENDOBEY
      .
ENDACTION
```

7.2 The Action KEYWORD Field

This field specifies the name by which the action is known to the person running the I-task.

The field is of the form:

```
KEYWORD  name
```

where `name` is a character string specifying a valid keyword. The same rules should be applied to action keywords as are applied to parameter keywords (see Appendix A.1).

Example

```
INTERFACE FILTASK
      .
   ACTION GET_FILTER
      KEYWORD FIND_FILTER
      OBEY
         NEEDS FILTER IN 'R', 'I', 'J'
         NEEDS DEAD_TIME RANGE 1.0, 5.5
      ENDOBEY
         .
   ENDACTION
      .
ENDINTERFACE
```

This field is used to specify the word that the user uses to communicate with a particular action. Thus the user could specify:

```
ICL> send filtask obey find_filter
```

As can be seen, this completes the separation between the program's view of actions and the user's view of them. It is possible for the programmer to re-write the program using completely different action names, but the user command could be kept the same by just changing the ACTION statements in the Interface File. Similarly the user's view of the program can be changed by just changing the action KEYWORD fields.

If the KEYWORD field is not specified for an action, then the system will use the action name as the keyword.

7.3 The HELP Field

This field is used to specify some help text for an action. No current user interfaces make use of this.

The field is of the form:

```
HELP help-text
```

where `help-text` is a character string giving help information about the action.

Example

```
ACTION GET_FILTER
   KEYWORD FIND_FILTER
   HELP 'This causes the filter to be selected'
   OBEY
      NEEDS FILTER IN 'R', 'I', 'J'
      NEEDS DEAD_TIME RANGE 1.0, 5.5
   ENDOBEY
      .
ENDACTION
```

8 ENTRIES FOR MENUS

There are two fields which are provided for possible future menu-style user interfaces. These can be used in both ACTION and PARAMETER specifications and are:

```
MENU menuname
MENUCOORDS xcoord,ycoord
```

where `menuname` is a character string specifying the name to be shown on the screen, and `XCOORD` and `YCOORD` are integers giving the coordinates for the position of the menu name. These quantities are not used by the parameter system, but merely stored.

A Parsing the Interface File

Interface File parsing is case-insensitive, the file is interpreted as a sequence of *tokens*, where a token is a sequence of characters which are either all alphanumeric (with . ; : + - ( ) _ [ ] " ' < > being honorary alphanumerics) or all non-alphanumeric (*i.e.* anything other than alphanumeric and 'white'). Tokens are thus terminated by anything of the opposite class or by a 'white' character. For this purpose, 'white' characters are space, tab, newline, comma or non-printable. Apart from their role as delimiters, 'white' characters are not significant.

The one exception to this is that, as a special case, a token may be a quoted string, *i.e.* a character string consisting of a set of characters enclosed in single quotes. The first quote must be the first character of the token and the token is terminated by the next isolated (*i.e.* not '') quote or the end of line. Thus two consecutive quotes in a quoted string will be interpreted as a single quote to be contained in the string.

Tokens may be one of:

- A reserved word. The reserved words are:

  ACCESS, ACTION, ASSOCIATION,
  CANCEL,
  DEFAULT,
  ENDACTION, ENDCANCEL, ENDINTERFACE, ENDMESSAGE,
  ENDMONOLITH, ENDOBEY, ENDPARAMETER, EPATH,
  HELP, HELPKEY, HELPLIB,
  IN, INTERFACE,
  KEYWORD,
  MENU, MENUCOORDS, MESSAGE, MONOLITH,
  NEEDS,
  OBEY,
  PARAMETER, POSITION, PPATH, PROGRAM, PROMPT, PTYPE,
  RANGE,
  TEXT, TYPE,
  VPATH

- A number constant (a valid Fortran number).
- A character constant (quoted string).
- A logical constant (Y, YES, T, TRUE, N, NO, F or FALSE, regardless of case).
- The constant !.
- A name. Anything which is not one of the above is classified as a name token. Name tokens can include a full file specification. Where such tokens are used as default values, case is preserved. Other names, such as parameter names and keywords are converted to upper case.

Fields (but not tokens) may be split across lines if necessary.

Everything following # on a line (except within a character string) will be ignored. Thus # can be used to introduce comments.

A.1 Parameter, Action and Keyword Names

The question of what is a valid parameter, action or keyword name is rather complicated. It is affected by Interface File parsing, the parameter system, HDS and the particular user interface in use. Appendix E gives the maximum lengths allowed and, to avoid problems, it is recommended that names begin with a letter and continue with *true* alphanumeric characters or underscore.

Parameter, action and keyword names are converted to upper case for storage. If they are displayed, in prompts or error messages *etc.*, they will appear in upper case.

A.2 Error Reporting

If an error is detected during the Interface File compilation phase, it is reported and, in most cases, the system is set into a state which is likely to enable it to continue. No further errors will be reported until a token acceptable in the new state is found. At the end of compilation a message giving the number of errors found will be reported. For some errors, where recovery is unlikely, compilation will stop immediately.

Unless compilation is aborted, a program reading the Interface File at run time, will go on and attempt to run with the information it has found and `compifl` will go on to write the `.ifc` file which could be read by the program later. In both cases the information obtained by the program will probably be incomplete and result in some unexpected defaults being used.

B Interface Module Search Path, ADAM_IFL

B.1 Introduction

By default, ADAM application programs attempt to find their Interface Module in the directory containing the executable file. A compiled (**.ifc**) file will be used in preference to a source (**.ifl**) file. To allow users to have different versions of the Interface Module for a given program without having multiple copies of the executable file, a search path mechanism has been implemented. This will be particularly useful for private tailoring of the Interface Modules of released software.

The search path is entirely the responsibility of the user – no value is defined in the standard system setup as there is a slight overhead in having one defined when it is not required.

If the search path is not defined, or an Interface Module is not found using it, the program will attempt to use the default.

B.2 Implementation for Unix

When a program is loaded, its name is obtained using system routine GETARG and ignoring any path component of the name. If environment variable ADAM_IFL is defined, it is assumed to specify a search path as a list of directory names separated by semi-colons. Each directory in turn is searched for a file with the same name as the program and with extension **.ifc** or, failing that, **.ifl**. If such a file is found, it is used as the Interface Module.

B.3 Implementation for VAX VMS

When a program is loaded, it attempts to translate the logical name ADAM_IFL using the tables defined by the SYSTEM logical name LNM$FILE_DEV (Normally PROCESS, JOB, GROUP and SYSTEM in that order).

ADAM_IFL may be a search path and, if it is defined, it is used as the filespec, together with a default filespec of 'filename.IF%' (where 'filename' is the filename of the executable image and '%' is any single character), in a call to LIB$FIND_FILE.

If such a file is found, the type is checked and, if the type is not .IFC or .IFL, the routine continues searching until no more files matching the specification are found. *Note that a .IFC file will be found before a .IFL file in any given directory.*

Notes:

(1) It is recommended that, if required, ADAM_IFL be defined as a JOB logical name. It cannot be a PROCESS logical name if the program is to be run in a subprocess, and GROUP logical names may cause confusion as they remain set between sessions.

(2) If ADAM_IFL includes filenames, they will override the name of the executable image.

C Parameter Specification for Output Parameters

Users whose programs have called a PAR_PUT routine are often surprised to be prompted for 'a parameter value'. This appendix is an attempt to clarify what is happening and what to do about it.

When the parameter system is asked for the value of a parameter of primitive type, the question it asks of the user via the Interface File) is not actually "What is the value of this parameter?" but "Where is the value of this parameter stored?". If the parameter system is provided with an HDS object name, the correspondence between the question and the answer is obvious. However, if a primitive constant is provided, the correspondence is not so clear. What is happening is that providing a primitive constant actually means "Store the given value in the program's parameter file[9], creating a component of appropriate name, type and size if necessary, and tell the parameter system where it is.". The program side of the parameter system then gets the value from where it is stored and delivers it to the program.

When a parameter value is to be output, the parameter system asks exactly the same question. Remembering that the prompt is actually for a location and not a value, the reason for the prompt is clearer.

It may be that output is required into some specific HDS object in which case its name may be specified in the normal way. (The object must exist and be of appropriate type and size.)

More often, the user just wants the parameter value to be put into the program's parameter file (possibly to update the GLOBAL parameter file eventually). There are various ways of specifying the VPATH in order to obtain this effect.

It is possible to be prompted and respond with a primitive value of an appropriate type. This will create a parameter file component of the correct type, in the same way as when a parameter value is 'got', and tell the system that the parameter file is to be used for the 'put'. (The actual value doesn't matter as it will be overwritten by the 'put'; the size and shape of the component will be changed if necessary. However, it is more efficient to specify the correct size if possible.)

If you do not want to be prompted for the parameter, the simplest way is to use the DEFAULT field.

```
PARAMETER parametername
      .
   ACCESS WRITE
   VPATH DEFAULT
   DEFAULT value
      .
ENDPARAMETER
```

where `value` is a primitive value of an appropriate type. Again the size does not matter (apart from efficiency), it will be changed if necessary.

[9] Or, in the case of scalar internal parameters, in memory

D Help Files for Multi-line Help

On Unix, the multi-line parameter help system uses the Starlink Portable HELP System (see SUN/124). If the specified file name has no extension or extension `.shl` or `.hlb`, `.shl` is assumed; anything else is an error.

On VMS, either Portable HELP or the VMS help system may be used. If the given filename has extension `.shl`, Portable HELP is used: if no file extension, or `.hlb`, is specified, the system will first look for a file with the given filename and extension **.shl**. If the file is found it will be used with the Portable Help System. In all other cases, the VMS help system is used.

In the interests of allowing the same Interface Modules to work on both VMS and Unix, both systems will handle names starting with (or wholly) environment variables (logical names) in either system's style.

E.g.

```
$environment_variable/relative_pathname
```

or

```
logical_name:filename
```

E Parameter System Constants

The common blocks used in ADAM application programs to hold information read from the Interface Module and required by the parameter system, impose certain limits on the size and number of parameters names, defaults and constraints *etc.* Further constraints are imposed by workspace requirements for the Interface File parsing system.

Current values for important limits are published here, together with the symbolic names used for the constants where appropriate.

Maximum Length of Names (characters)

Parameter names	15	SUBPAR__NAMELEN
Parameter keywords	15	SUBPAR__NAMELEN
Action and program names	15	SUBPAR__NAMELEN
Action keywords	15	SUBPAR__NAMELEN
Menu names	15	SUBPAR__NAMELEN
Monolith names	80	

Maximum Lengths of Strings (characters)

Length of parameter value string	132
Length of PROMPT string	80
Length of HELP string	132
Length of HELPKEY string	132
Length of action HELP string	132

Maximum Numbers of Items

Number of parameters	1500	SUBPAR__MAXPAR
Number of actions	300	SUBPAR__MAXACT
Number of menus	300	SUBPAR__MAXACT
Number of command line parameters	50	
Size of NEEDS storage	300	SUBPAR__MAXNEEDS
Size of storage for constraints/defaults	500	SUBPAR__MAXLIMS

Note: There are SUBPAR__MAXLIMS positions of each type available for constraints/default storage. One position of the relevant type is used for every value given in IN, RANGE and DEFAULT fields or specified as a dynamic default by the program.

Maximum Parsing Workspace

Length of line (characters)	132	PARSE__BUFSIZ
Length of token (characters)	132	PARSE__BUFSIZ
Number of tokens on a line	32	

F Obsolete Fields

Several fields which were relevant to the original Starlink Software Environment (SSE) are not used with ADAM. The fields are described here so that the ideas behind them may be preserved and because they may still be found in old software and documentation.

F.1 The Program Specification

The ADAM application program to be run is determined by the user interface when the appropriate command is issued. The program then finds the corresponding Interface Module (see Appendix B). The SSE worked in a different way – the user interface found the Interface Module first and the Interface Module defined which executable image was to be run. The Program Specification fields controlled the name of the program to be executed when a request to run an application is received, and where to find the executable image.

There are 2 Program Specification fields: PROGRAM and EPATH.

F.1.1 The PROGRAM field

This Field Specification gives the name of application program to be executed when the user invokes this Interface Module. It is still accepted by the system but has no effect.

The field is of the form:

```
PROGRAM progname
```

where `progname` is a character string (or reserved word) which is the name of the executable image without the '.EXE' file extension.

Example

```
PROGRAM 'DECONV'
```

F.1.2 The EPATH field

This field specification provided the search path for locating the directory in which the executable image of the application program was to be found. It is still accepted by the system but has no effect.

The field is of the form:

```
EPATH dpath
```

where `dpath` is a character constant listing the order of search through subdirectories. The directory specifications in the path are separated by semi-colons. A null path indicates the current directory. Note that logical names must end with a colon.

Example

```
EPATH 'DISK$USER1:[SLW.JUNK];SAI_LAPPLIC:'
```

F.2 The RPATH field

This field provided a search path for data system object names. It is no longer accepted by the system. The field specifier was of the form:

```
RPATH resolution-path
```

where resolution-path was a character string giving a list of resolution path specifiers separated by semi-colons. Each specifier was a data-system object name.

Example

```
INTERFACE DISP
      .
   PARAMETER DISPLAY
         .
      RPATH 'TEMP.DEVICES;"SAI_USER:DEVICES";"LSAIDIR:DEVICES"'
         .
   ENDPARAMETER
      .
ENDINTERFACE
```

Then if the parameter DISPLAY was given the value ARGS, the system attempted to use the object TEMP.DEVICES.ARGS then, failing that SAI_USER:DEVICES.ARGS and so on. (Note that ADAM no longer uses a 'DEVICE dataset' object when opening graphics devices.)

G Changes in this Document

G.1 SUN115.2

Apart from the format, there has been little change in this document. Some points are clarified and references to out-of-date documents replaced or removed. The move from DCL to Unix is also reflected.

The main significant changes are as follows:

The DEFAULT field (Section 4.8 Quoted strings may now be given as defaults for non-primitive types – the string within the quotes will be taken as the 'name'.

NEEDS (Section 7.1) The changes introduced with ADAM V2.0 whereby the only effect of NEEDS is to define command line positions, are reflected. There are corresponding changes to the RANGE and IN descriptions.

RANGE (Section 4.11) The interaction between the dynamic minimum/maximum system and the RANGE is described.

ACCEPT (Sections 4.10 and 4.14) The ACCEPT keyword now only affects parameters for which a prompt would otherwise have been issued. Parameters may also now be forced to ACCEPT by the `keyword=\` parameter specifier or by typing '\' in response to a prompt.

Suggested Values The term *prompt value* has been replaced by *suggested value* which is now the preferred term.

Help Files Appendix D has been added to give details of the filenames which may be given with the HELP, HELPKEY and HELPLIB fields.

Output Parameters Appendix C has been altered to describe the simpler method of avoiding prompts for output parameters now that the type, shape and size of parameter file components will be changed automatically if necessary when a parameter value is 'put'.

G.2 SUN/115.3

Minor re-formatting and typo correction.