Programmer's Reference Manual

PANOS

Programmer's Reference Manual

PANOS

PART NO 0410, 012
ISSUE NO 1
JULY 1985

The product described in this manual and products for use with it, are subject to continuous developments and improvement. All information of a technical nature and particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

In case of difficulty please contact your supplier. Deficiencies in software and documentation should be notified in writing, using the Acorn Scientific Fault Report Form to the following address:

Sales Department
Scientific Division
Acorn Computers Ltd
Fulbourn Road
Cherry Hinton
Cambridge
CB1 4JN

All maintenance and service on the product must be carried out by Acorn Computers' authorised agents. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

Published by Acorn Computers Limited, Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN.

Within this publication the term BBC is used as an abbreviation for the British Broadcasting Corporation.

NOTE: A User Registration Card is supplied with the hardware. It is in your interest to complete and return the card. Please notify Acorn Scientific at the above address if this card is missing.

ISBN 0 907876 39 0 Acorn Scientific

Contents

Introduction

This document describes the programmer's interface to Panos, the operating system for Acorn Cambridge Series computers. Panos rests on the low level machine support presented by Pandora, and provides a runtime system to support a range of high level languages.

The user gains access to the functionality provided by Panos via:

- A command line interpreter (CLI)
- A collection of utility programs
- The runtime library

The Panos command line interpreter, utility programs, and other user-interface related matters are described in the *Panos Guide to Operations.*

The majority of this document is taken up by a description of the Panos library. Each chapter deals with a particular module which contains one or more procedures of a given class, e.g. random numbers, command handling.

Appendix A of this manual lists the error codes associated with the user library procedures.

All procedures are described using a pseudo-language notation which lists the number and type of the parameters and results. Parameters and results are passed according to the rules described in the document *Panos Technical Reference Manual.* An informal introduction to this pseudo-language is given in chapter 1.

The following convention is observed:

Numbers not in decimal are prefixed by their base, for example 16_1A is decimal 26; -2_1010 is -10 in decimal.

1 The user library

All explicit communication with Panos from a user program is via the user library procedures. There are two ways by which Panos informs the user of errors; either the library procedure returns an error status (which should be checked explicitly by the caller) or an error exception is signalled. All library routines are provided in two versions, with the variant that generates an exception on error having its name prefixed by 'X'.

The library of Panos is divided up into several modules (see 'Acorn 32000 object format specification' in the *Panos Technical Reference Manual* for a description of the term 'module'). Each group of procedures described in the following chapters resides in a separate module. The module name is given at the bottom of each page for procedure descriptions.

Procedures are described in terms of a pseudo-programming language. The method of interfacing with real languages such as FORTRAN 77 and Pascal depends on the procedure calling system of each language. Most languages under Panos conform to the Acorn inter-language calling standard. In Acorn 32000 ISO Pascal, for example, a Panos library procedure can be accessed by the IMPORT directive.

For example, the method for importing the procedure SetKnownCommandsPath is:

```
type
   string=packed array[1..15] of char; { bound specified as required }
....
....
import function SetKnownCommandsPath(path:string;len:integer):integer
....
status := SetKnowncommandsPath('$.Panoslib, @ ',13)
....
```

Full details of the inter-language calling standard and the Acorn Object Format produced by Acorn compilers are given in the *Panos Technical Reference Manual.* Explanation of how this maps into a particular language's calling system is given in that language's Reference Manual.

The procedures in this manual are described by the following syntax:

```
<procedure description> ::= <procedure name>(<parameter list>);
                            <result list>
<parameter list>        ::= | <parameter> <parameter list>
<result list>           ::= | <result> <result list>
<parameter>             ::= <parameter type>:<parameter name>
<result>                ::= <result type>:<result name>
<parameter type>        ::= STRING | <base type> | <base type>REF
<result type>           ::= STRING | <base type>
<base type>             ::= INTEGER | CARDINAL
                            | RECORD(<format name>)
                            | ADDRESS
                            | HIDDEN
```

These types are described in the inter-language calling standard section of the *Panos Technical Reference Manual* with HIDDEN being 32-bit raw binary.

2 Errors

This chapter describes the facilities provided for error handling; all of the procedures reside in moduleError . Many of the procedures in the Panos library return a 32-bit status code. On error the top bit of the status code is set, so the number is always negative. The other 31 bits are divided into fields which provide information about the error type, in which module it occurred and so on.

When an error occurs in a system module, it will call the procedure SetErrorInformation. This takes an error code and assigns an 'information string' to this error. The error number is then returned to the caller.

The calling program uses GetErrorMessage to convert the returned error number into three information strings: the message, the name of the system facility which detected the error and the name of the system facility which was initially called by the user.

Here is a typical sequence of events which might result in the error-handling procedures being called:

A program calls the procedure GetDateStamp to find the date at which a named file was created. The name is supplied as the string parameter 'DFS::0.$.prog1'. However, the named file does not exist on the filing system, so an error is generated. Suppose the basic error message is:

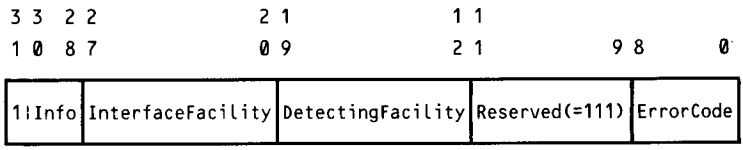
File % not found

The system will set the information for the error to the filename, i.e. 'DFS::0.$.prog1'. Thus when the user calls GetErrorMessage using the returned error code, the three strings returned will be:

```
BBC
File
File DFS::0.$.prog1 not found
```

where BBC is the detecting facility (i.e. the module which discovered that the file did not exist), File is the interface facility (i.e. the module that the user called in the first place) and 'File DFS::0.$.prog1 not found' is the error message with the error information substituted.

All error codes generated by the Panos system and associated software are 32-bit values with the most significant bit set. They are divided into a number of fields:



The structure of this error code has been designed such that a simple user program can return a small negative number (range -1 to -512) to denote an error condition.

Info

This field describes whether any additional information is available for this error (see GetErrorInformation). The values in the field have the following meanings:

0	None; the error has already been reported.
1-5	Information available: use the value of the 32-bit errorcode as a handle to GetErrorInformation to obtain it.
6	Used when error is being signalled (see module 'Handler') to denote the passing of an associated error buffer.
7	None.

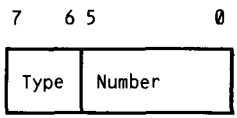
InterfaceFacility

This gives the code of the facility called by the user. This may be converted to a string using GetErrorMessage. The table given below lists the facility names also.

DetectingFacility

Gives the code of the facility which detected the error. This is also decoded by GetErrorMessage.

For both facility fields the code is structured as below:



The type field's two bits are:

Type 00 means that Number is a Panos facility as follows:

0	32000 Hardware exception
1	Data conversion
2	Store
3	IO
4	Loader
5	Random number generation
6	Time and date
7	Condition handling.
8	Event handling
9	Environment variables
10	Program control
11	Pandora
12	BBC
13	Argument decoding (error in keystring format)
14	Argument decoding (error in user parameter string)
15	File
16	Reserved
17	Command interpreter
18	Error handling
19	Pattern matching
20-63	Reserved

For Number 0 the ErrorCode (bits 0-8) is the 32000 Hardware Exception code (see the *Instruction Set Reference Manual* for details).

Type 2_01 is reserved.

Type 2_10 implies that Number is a Language Code (see 'Acorn 32000 Object Format specification' in the *Panos Technical Reference Manual*). The ErrorCode is a compiler error.

Type 2_11 implies that Number is a Language Code. The ErrorCode is a run-time error.

ErrorCode

See above.

Note: Facility 16_FF is reserved for user programs.

The system procedure GetErrorMessage is provided to convert an error code into a textual message. The mapping between an error code and its corresponding text message is controlled by the system error file.

The error file is made up of a sequence of records separated by newline (LF) characters. Each record has the format

<ff> <ee> <skeleton error message>

where <ff> is a two digit facility number (base 16), e.g. 0F for File, and <ee> is the facility error code.

The <skeleton error message> is a printable message containing % characters where substitution of error information is required. The error information set for an error is made up of % separated fields.

When GetErrorMessage is building the message string it will substitute fields in the message skeleton from those in the corresponding position in the error information. If insufficient fields are provided in the error information then the value <unknown> will be used.

For example:

Skeleton	Error % on stream %
Information	87%12
Produces	Error 87 on stream 12
Skeleton	File % not found on FS %
Information	$.file1
Produces	File $.file1 not found on FS <unknown>

2.1 GetErrorMessage

GetErrorMessage(INTEGER:error);
INTEGER:Result
STRING:DetectingFacility
STRING:InterfaceFacility
STRING:Error message

XGetErrorMessage(INTEGER:error);
STRING:DetectingFacility
STRING:InterfaceFacility
STRING:Error message

Action

Decodes the supplied error number and returns three strings describing:

The System Facility which detected the Error,
The System Facility which was called by the user and
A text string describing the error.

Any additional information about the error is merged into the error message.

Call

Error The error number.

Return

Result $>=0$, operation succeeded.
<0, operation failed (result = Error Code).

2.2 SetErrorInformation

SetErrorInformation(INTEGER:Error
STRING:Information);
INTEGER:Result

XSetErrorInformation(INTEGER:Error
STRING:Information);
INTEGER:Result

Action

This caches the information string and sets the 'Info' field of the given error. The modified error can be used at a later stage as a parameter to GetErrorInformation which will endeavour to return the information string. For this to be successful only the Interface Facility in the resultant Error Code may be changed.

Call

Error	An Error Code complete except for Interfacing Facility, which may be modified at a later stage.
Information	The information string to be associated with the error, e.g. 'DFS::0.$.fred%13'.

Return

Result	Can be used both as an error code and as a handle to get back the information.

2.3 GetErrorInformation

GetErrorInformation(INTEGER:Error);
INTEGER: Result
STRING: Information

XGetErrorInformation(INTEGER:Error);
STRING Information

Action

Endeavours to return any additional information associated with a system Error Code.

Call

Error The Error Code.

Return

Result $>=0$, Operation successful, information contains the additional information.
<0, Operation failed (= Error Code).

3 Argument decoding

When a program is called, its name may be followed by a list of arguments for use as parameters, e.g. source and object filenames for compilers, file specifications for filing system utilities etc.

This section describes the Panos procedures which can perform decoding of command line arguments. By utilising these procedures, all applications running under Panos can provide a uniform command interface to the user. The procedures all reside in moduleDecodeArg.

Decoding is performed by passing Panos a keystring which describes the format of arguments which the program expects. The procedure DecodeInit takes the keystring and processes the argument string accordingly. Decoded parameters may then be accessed by calling various other procedures, e.g. GetStringArg. This process is illustrated in figure 1.

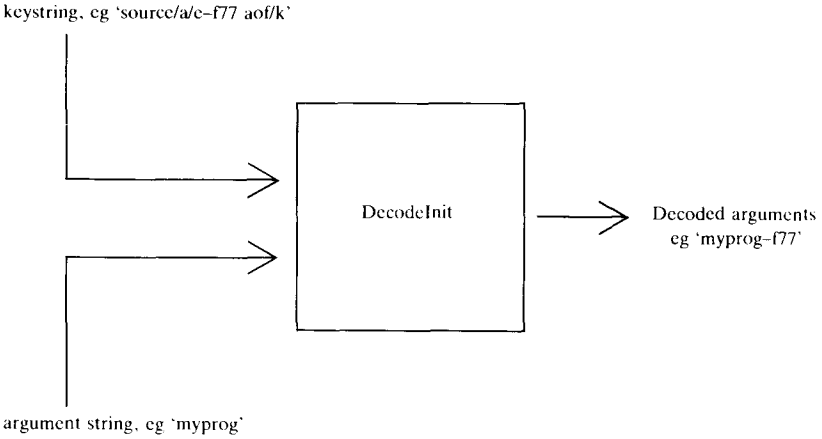


Figure 1 Argument Decoding

A keystring is a sequence of keywords, which are qualified by control characters called option specifiers (e.g. /a and /e in the example above). These determine the type of keyword, and the number and type of arguments which may be associated with it.

When all of the arguments have been obtained by calls to the Get...Arg procedures, the application program should terminate the decoding process cleanly by calling DecodeEnd.

The keystring

This section describes the format of the keystring in detail. As stated above, it is a list of keywords (separated by spaces or commas) which may be qualified by option specifiers.

Associated with each keyword may be a default argument list. This is used if the user does not supply any arguments on the command line for that keyword.

Keyword name

A keyword name must begin with a letter and can contain letters or digits (Underscore, '_', is treated as a lower case letter in keywords). The case of the keyword in the keystring is used to permit controlled abbreviation of the use of the keyword in the supplied argument string. (The actual case of the keyword in the supplied argument string is irrelevant.)

Abbreviation of a keyword when given in the argument string is possible if the keyword in the keystring ends in a sequence of lower case letters. Only lower case letters in the keystring keyword name may be truncated from the argument string keyword name. An abbreviation is not permitted if it is an otherwise legal truncation of more than one keyword.

An example is

'NAme' in the keystring may be matched from the argument string by '-NAME' (or '-NAM', and '-NA' provided this is unambiguous). '-N', will not match. A '-NAME' will match in preference to abbreviations of any longer keyword names (i.e. is not ambiguous with keystring 'NAme NAMEList').

Keyword names may be aliased by separating each alias with an equals sign, e.g. 'FROM = INput' will match '-FROM', '-INPUT', '-INPU', '-INP', and '-IN'.

Option specifiers

There are three classes of option specifier:

1. Quantity option

This is used to indicate the number of arguments which may be associated with the keyword. There are three formats:

/<num> This specifies that at most <num> arguments may be associated with the keyword e.g. /2, or /12.

/=<num> This specifies that exactly <num> arguments must be supplied for the keyword. e.g. /=1, or /=4.

/? This specifies that any number of arguments can be supplied. A procedure (GetNumberOfValues) is supplied to discover how many arguments a user actually provided.

If no quantity option is supplied then /1 is assumed, i.e. keywords are expected to have at most one argument by default. Suppose a keystring contains the keyword INPUT/?. This means that a command line could contain the word -INPUT followed by any number of arguments to be associated with that keyword. The arguments should be separated by commas, so an example might be -INPUT fred,jim, sheila.

Having called DecodeInit with the appropriate keystring, the application program could discover how many arguments are associated with INPUT by calling GetNumberOfValues, and then read each one by calling GetStringArg.

2. Type option

This option indicates what type of arguments are expected to be associated with a given keyword. The possible options are:

/I This indicates that integer arguments will be used with the keyword. Examples are 128, 16_1A, -3024, +8_1764. Integer arguments may be read with GetIntegerArg.

/C This indicates that cardinal (positive integer) arguments will be used with the keyword. Examples are 128, 16_1A, 3024, 8_1764. Cardinal arguments may be read with GetCardinalArg.

/B This indicates that boolean arguments (True, False) will be used with the keyword. See also the keyword presence options. Boolean arguments may be read with GetBooleanArg.

/E[-ext] This means that the keyword's arguments are expected to be files residing on some filing system. /E arguments may be read with GetStringArg. GetStringArg checks that the name provided exists before returning and gives an error if it does not. In addition, an extension may be given which is automatically appended to filenames which don't have an extension already.

/R This indicates that the keyword's argument consist of the rest of the argument string. This keyword must be specified last in the keystring since once it is selected (even by default) other keywords have no effect. It is supplied to allow partial decoding of argument lines. /R arguments may be read with GetStringArg.

/L This indicates that the keyword's arguments consists of the long string from this keyword position to the next valid keyword. It does not include the leading or trailing white space. A quoted string will have the outermost quotes stripped. /L arguments may be read with GetStringArg.

If no type option is supplied then the key word has arguments of type STRING.

3 Keyword presence options

There are two options to control the necessity of giving the keyword name in the argument string, and three to control the detection of keyword names in the argument string.

/A This implies the keyword must have at least one argument (though the keyword itself need not occur in the argument string). The keyword cannot be used with a default argument list.

/K This means that the keyword can have arguments only if the keyword itself is also given.

/P This allows a keyword to have no arguments and implies /K. The function GetPresence returns TRUE if the keyword name was given in the argument string.

As an example consider the keystring 'list/P[vdu:]' supplied with different argument strings.

Argument	Presence	Number	Value
Empty	FALSE	1	no listing is wanted
'-list'	TRUE	1	vdu: list to the vdu:
'-list printer:'	TRUE	1	printer: list to printer:

Hence, GetPresence would be used to determine whether listing is wanted. Note that /P can cause ambiguity since it only sometimes takes an argument.

/N This implies /P and allows a '-NO' prefix to the keyword name. The function GetStateArg can be used to find the state of the rightmost use of the name. If the -NO prefix is given then no argument can be supplied and any arguments given to previous unprefixed uses of this name are ignored.

As an example consider the keystring 'list/N[vdu:]' supplied with different argument strings.

Argument	Presence	Number	State
Empty	FALSE	1	FALSE no listing is wanted
'-list'	TRUE	1	TRUE list to the vdu:
'-list printer:'	TRUE	1	TRUE list to printer:
'-nolist'	TRUE	0	FALSE no listing is wanted
'-nolist -list'	TRUE	1	TRUE list to the vdu:
'-list -nolist'	TRUE	0	FALSE no listing is wanted
'-list xx -nolist'	TRUE	0	FALSE no listing is wanted

Hence, GetStateArg would be used to determine whether listing is wanted. Note that '-nolist xx' is illegal since the -nolist cannot take an argument.

/S This is equivalent to /K/P/N/=0 and specifies a keyword which can have no arguments but can be detected in the arguments string. Examples are 'Identify', and 'HELP'.

Defaults

A keyword which can take arguments can have defaults supplied as a comma separated list in square brackets.

Assignment of arguments to keywords

This section describes how the user-supplied arguments are decoded using the programmer-supplied keystring.

Terms used are:

Argument group: An argument group is a list of comma-separated arguments associated with a keyword.

Argument string: An argument string comprises one or more argument groups.

Argument: An argument is a single string, filename, integer, cardinal, or boolean.

The rules for decoding are:

a The default order of assigning argument groups to their corresponding keywords is from left to right. Note that /K keywords are not eligible for default assignment and so the keyword is skipped.

b Argument groups qualified by the keyword name can be supplied in any order.

c A non-/K keyword, when specified, just changes the default keyword for assignment. It need not have an argument following the keyword.

d An item in the argument string beginning with a '-' must be immediately followed by a keyword name (possibly abbreviated) or a digit. The /L and /R keywords are treated specially and cause exceptions to this rule.

e An item that is enclosed in quotes ('') cannot be a keyword, it is always treated as an argument. A /L keyword causes an exception to this rule.

f If no keyword name is supplied and there is no default keyword then the argument group cannot be assigned. An error is generated.

g If the keyword has already received an argument group then the current argument group is assigned to this keyword after the existing arguments.

h The keyword names HELP and IDENTIFY are special, in that if all rules are not satisfied but HELP or IDENTIFY are supplied within both the keystring and the argument string then a special error number is returned. This allows users to type e.g. 'f77 -Help' and receive some help information.

3.1 ArgumentInit

```
ArgumentInit(STRING:KeyString
             BOOLEAN:InputWanted
             BOOLEAN:OutputWanted
             STRING:Identification
             PROCEDURE:HelpProcedure);
             INTEGER:Result
             HIDDEN:Handle

XArgumentInit(STRING:KeyString
              BOOLEAN:InputWanted
              BOOLEAN:OutputWanted
              STRING:Identification
              PROCEDURE:HelpProcedure);
              HIDDEN:Handle
```

Action

This procedure is provided to allow a simple program to behave in the standard manner for programs. A standard string is prefixed to the keystring, DecodeInit is called with the value of Program.Arguments() and then some standard operations are performed. The decoded information handle is then returned, and can be used with GetStringArg etc.

The prefix string contains the following keywords in the order given below.

ERRor/K[Error:]
: supplies the value for IO.SetErrorStream(IO.FindInput(value))

CONTrol/K[Control:]
: supplies the value for IO.SetControlStream(IO.FindInput(value))

IDentify/S
: If this argument is present on the command line then the value of Program.ProgramPath(), a colon, a space, and then the value of the Identification parameter, followed by a newline are output to ErrorStream.

HELP/S

If this argument is present on the command line then the HelpProcedure parameter is called, it should output help to the current ErrorStream. When it returns a Program.Stop(0) is performed.

Abandon/S

The required value can be obtained from

Program.SwitchRequired("Abandon", Handle, Default)

A value of True should cause the program to stop after the first error, if False then the program should keep going for as long as possible.

e.g. 'Copy a,b,c -to d -Abandon'

will not copy file 'c' if file 'b' is not found.

Confirm/S

The required value can be obtained from

Program.SwitchRequired("Confirm", Handle, Default)

A value of True should cause the program to ask the user for confirmation of the operation if this is reasonable. e.g. delete -confirm will ask for confirmation, but f77 -confirm will have no effect.

Force/S

The required value can be obtained from

Program.SwitchRequired("Force", Handle, Default)

A value of True should cause the program to take as much action as is reasonable to ensure that the requested task can be accomplished. e.g. delete -force will override the locked attribute on files.

The following two keywords are optionally included depending upon the values of the corresponding 'wanted' boolean arguments.

INput = FROM[Input:]

supplies the value for IO.SelectInput(IO.FindInput(value))

OUTput = TO[Output:]

supplies the value for IO.SelectOutput(IO.FindOutput(value))

Call

KeyString
: the keys to decode the arguments.

InputWanted
: If true then add 'INput=FROM[Input:]' to the keystring.

OutputWanted
: If true then add 'OUTput=TO[Output:]' to the keystring.

Identification
: the string to be output as identification.

HelpProcedure
: a procedure without parameters to produce help.

Return

Result
: $>= 0$, decode successful and Handle refers to the decoded arguments and the standard operations have been performed.
< 0, operation failed (= error code). See Appendix A.

Example

A very simple program which takes no arguments could call ArgumentInit and give it parameters:

KeyString = ""

Identification = "Sound bell version 1.00"

HelpProcedure = HelpProc { A simple procedure to say that this program just sounds the bell }

InputWanted = false

OutputWanted = false

This program would then behave in the accepted manner.

3.2 DecodeInit

DecodeInit(STRING:KeyString
STRING:ArgString);
INTEGER:Result
HIDDEN:Handle

XDecodeInit(STRING:KeyString
STRING:ArgString);
HIDDEN:Handle

Action

Decode the supplied argument string using the keystring and set up the results so that the values of arguments can be obtained by the variants of GetArg. The keystring is a proforma for the arguments expected.

Call

KeyString	the keys to decode the arguments.
ArgString	the argument string.

Return

Result	$>= 0$, decode successful and Handle refers to the decoded arguments. < 0, operation failed (= error code). See Appendix A.

3.3 GetStringArg

GetStringArg(STRING:ArgumentName
CARDINAL:Index
HIDDEN:Handle);
INTEGER:Result
STRING:Arg

XGetStringArg(STRING:ArgumentName
CARDINAL:Index
HIDDEN:Handle);
STRING:Arg

Action

Obtain the value of the named STRING argument from the decoded argument list. 'Index' identifies which of the possible N values is required. For example Index = 1 corresponds to the first value associated with the ArgumentName.

Call

ArgumentName	key name for the argument.
Index	identifies which of the possible N values is required.
Handle	As obtained from DecodeInit.

Return

Result	> = 0, operation successful (arg contains the value). < 0, operation failed (= error code).

3.4 GetStateArg

GetStateArg(STRING:ArgumentName
HIDDEN:Handle);
INTEGER:Result
BOOLEAN:Arg

XGetStateArg(STRING:ArgumentName
HIDDEN:Handle);
BOOLEAN:Arg

Action

This can only be used on /N or /S arguments and returns TRUE if the rightmost use of the keyword was without the -NO prefix. It returns FALSE if the keyword was not specified in the argument string or the rightmost use was with a -NO prefix.

Call

ArgumentName	key name for the argument.
Handle	As obtained from DecodeInit.

Return

Result	$>=0$, operation successful (arg contains the value). <0, operation failed (= error code).

3.5 GetBooleanArg

```
GetBooleanArg(STRING:ArgumentName
              CARDINAL:Index
              HIDDEN:Handle);
              INTEGER:Result
              BOOLEAN:Arg

XGetBooleanArg(STRING:ArgumentName
               CARDINAL:Index
               HIDDEN:Handle);
               BOOLEAN:Arg
```

Action

Obtain the value of the named boolean argument from the decoded argument list.

Call

ArgumentName	Key name for the argument.
Index	Identifies which of the possible N values is required.
Handle	As obtained from DecodeInit.

Return

Result	$>=0$, operation successful (arg contains the value). <0, operation failed (= error code).

3.6 GetIntegerArg

GetIntegerArg(STRING:ArgumentName
CARDINAL:Index
HIDDEN:Handle);
INTEGER: Result
INTEGER: IntegerArg

XGetIntegerArg(STRING:ArgumentName
CARDINAL:Index
HIDDEN:Handle);
INTEGER: IntegerArg

Action

Obtain the value of the named integer argument from the decoded argument list.

Call

ArgumentName	Key name for the argument.
Index	Identifies which of the possible N values is required.
Handle	As obtained from DecodeInit.

Return

Result	$>=0$, operation successful (IntegerArg contains the value). <0, operation failed (= error code).

3.7 GetCardinalArg

```
GetCardinalArg(STRING:ArgumentName
               CARDINAL:Index
               HIDDEN:Handle);
               INTEGER:Result
               CARDINAL:CardinalArg

XGetCardinalArg(STRING:ArgumentName
               CARDINAL:Index
               HIDDEN:Handle);
               CARDINAL:CardinalArg
```

Action

Obtain the value of the named Cardinal argument from the decoded argument list.

Call

ArgumentName	Key name for the argument.
Index	Identifies which of the possible N values is required.
Handle	As obtained from DecodeInit.

Return

Result	$>=0$, operation successful (CardinalArg contains the value). <0, operation failed(= error code).

3.8 GetNumberOfValues

GetNumberOfValues(STRING:ArgName
HIDDEN:Handle);
INTEGER:Result
CARDINAL:Number

XGetNumberOfValues(STRING:ArgName
HIDDEN:Handle);
CARDINAL:Number

Action

Returns the number of values supplied for the keyword ArgName.

Call

ArgName Key name for the argument.

HAndle as obtained from DecodeInit.

Return

Result $>=0$, operation successful (Number contains the number of values supplied).
<0, operation failed (= error code).

3.9 GetPresence

GetPresence(STRING:ArgName
HIDDEN:Handle);
INTEGER:Result
BOOLEAN:Present

XGetPresence(STRING:ArgName
HIDDEN:Handle);
BOOLEAN:Present

Action

Returns True if the ArgName was specified in the argument string.

Call

ArgName	Key name for the argument.
Handle	As obtained from DecodeInit.

Return

Result	$>=0$, operation successful. <0, operation failed (= error code).
Present	True if the ArgName was present.

3.10 Substitute

Substitute(STRING:String
HIDDEN:Handle);
INTEGER:Result
STRING:SubstitutedString

XSubstitute(STRING:String
HIDDEN:Handle);
STRING:SubstitutedString

Action

Scans the 'String' searching for words bracketed between < and > and substituting each <Word> for either:

a the string equivalent value of the keyword name <Word> associated with the given Handle. The special case <-Word> is treated differently. Word must be a keyword of type /E. The value substituted is the argument with any extension removed.

b the value of global string Word.

Note: This is the procedure used by the Panos command interpreter to substitute parameters in input lines.

Call

String — The string containing <word>'s to be substituted.

Handle — As obtained from DecodeInit.

Return

Result — >=0, operation successful (= number of substitutions).
<0, operation failed (= error code).

SubstitutedString

the Resulting string after substitution.

3.11 DecodeEnd

DecodeEnd(HIDDEN:Handle);
INTEGER:Result

XDecodeEnd(HIDDEN:Handle)

Action

Indicate end of argument decoding. All Heap memory used by Panos for decoding purposes is released.

Call

Handle As obtained from DecodeInit.

Return

Result $>=0$, operation successful.
<0, operation failed (= error code).

4 Data conversion

In order to support the Acorn standard Panos makes available several procedures for conversion between data representations. These procedures reside in moduleConvert . For consistency between programs, it is highly desirable that these procedures should be used to perform conversions.

The standard stipulates that numbers not in decimal are prefixed by their base and an underscore character ('_'). The base is a decimal value and is an integer value in the range 2..36. For example:

16_1A	is decimal	26,
56	is decimal	56,
8_17	is decimal	15.

Numbers may optionally be preceeded by the + or - signs. For example:

-16_1a	is decimal	-26,
+8_17	is decimal	15,
-2_111	is decimal	-7.

Any other string representation of numbers is rejected.

4.1 StringToInteger

StringToInteger(STRING:SourceString);
INTEGER:Result
INTEGER:IntegerResult

XStringToInteger(STRING:SourceString);
INTEGER:IntegerResult

Action

Converts a string into an integer.

Call

SourceString — The string to be converted in standard format. For example, 2_10101, -2_101011, +2_101011 24, 16_ABCD. A leading - sign indicates a negative value. The range of integers is -(2**31)..(2**31)-1. All other input is illegal.

Return

Result — > = 0, operation successful and IntegerResult is the integer result.
< 0, operation failed (= error code).

4.2 StringToCardinal

StringToCardinal(STRING:SourceString);
INTEGER: Result
CARDINAL:CardinalResult

XStringToCardinal(STRING:SourceString);
CARDINAL:CardinalResult

Action

Converts a string into a cardinal.

Call

SourceString — The string to be converted in standard format. For example, 2_10101, 24, 16_1ADD. Cardinals range from 0..(2**32)-1. All other input is illegal.

Return

Result — > = 0, operation was successful and CardinalResult is the result.
< 0, operation failed (= error code).

4.3 IntegerToString

IntegerToString(INTEGER:Number
CARDINAL:TheBase);
INTEGER:Result
STRING:ResultString

XIntegerToString(INTEGER:Number
CARDINAL:TheBase);
STRING:ResultString

Action

Converts an integer into a string using given base.

Call

Number integer to be converted.

TheBase conversion base, in the range 2..36.

Return

Result $>= 0$, operation successful ResultString is the result.
< 0, operation failed (= error code).

4.4 CardinalToString

CardinalToString(CARDINAL:Number
CARDINAL:TheBase);
INTEGER:Result
STRING:ResultString

XCardinalToString(CARDINAL: Number
CARDINAL:TheBase);
STRING:ResultString

Action

Converts a cardinal into a string using given base.

Call

Number	Cardinal to be converted.
TheBase	Conversion base, in the range 2..36.

Return

Result	> = 0, operation successful, ResultString is the result. < 0, operation failed (= error code).

4.5 BooleanToString

BooleanToString(BOOLEAN: bool);
INTEGER:Result
STRING:ResultString

XBooleanToString(BOOLEAN: bool);
STRING:ResultString

Action

Convert a boolean value into its string representation. The string representations used are 'True' and 'False'.

Call

bool The boolean value to be converted.

Return

Result $>=0$, Operation successful and ResultString is the result.
<0, Operation failed (= error code).

4.6 StringToBoolean

StringToBoolean(STRING: sourcestring);
INTEGER: Result
BOOLEAN: BooleanResult

XStringToBoolean(STRING: sourcestring);
BOOLEAN: BooleanResult

Action

Converts a string repesentation into a boolean.

Call

sourcestring	The string to be converted. (The representations accepted are 'true' and 'false', in any mixture of upper and lower case letters.)

Return

Result	$>=0$, Operation successful and BooleanResult is the value. <0, Operation failed (= error code).

5 Store allocation

The Store allocation part of Panos manages the memory available to the user program. Storage allocation/deallocation is carried out by the use of procedures in moduleStore .

The lowest part of memory (up to the first 64K bytes) contains the module tables. An area of memory, the 'module heap' is maintained so that extra module information may be added and deleted as required by the loader part of Panos.

The main heap comprises all the rest of the free store. The upper limit to the growth of the main heap is called HeapEnd. HeapEnd is normally determined by the position of the stack pointer: it is usually just below the stack pointer (a 'safety margin' is included), but it may be fixed at a user specified address if required. The calculation of Heapend may be returned to its default mechanism by a call to ResetHeapEnd.

When memory is returned to the allocator it is coalesced with any neighbouring free memory. This increases the likelihood of meeting future demands for large amounts of contiguous memory.

The allocator exports a variable, CurrentHeapEnd, which contains the address of the first free byte above all claimed heap data. This exported data item is for stack checking by user programs and should only be read.

For the convenience of the user, the allocator maintains a tag, the Store Tag, which can be set and read using a supplied procedure. Each time a block of memory is allocated it is tagged, the user may either deallocate memory blocks individually or choose to deallocate all blocks with a particular tag. Blocks tagged with tag = 0 cannot be deallocated in a group (see DeallocateGroup library call below).

5.1 Allocate

Allocate(INTEGER:Size);
INTEGER:Result
ADDRESS:BlockPointer

XAllocate(INTEGER:Size);
ADDRESS:BlockPointer

Action

Allocate a block of store and tag with the current value of StoreTag (as set by last call to SetStoreTag).

Call

Size	The number of bytes required. If size is positive then at least size bytes must be allocated. If size is negative then |size| represents an upper bound on the amount required. The argument value Size < 0 is not available for XAllocate. Size may not take the value 0.

Return

Result	$>=0$, operation successful (= number of bytes allocated) and BlockPointer points to the start of the block which will be four-byte aligned. <0, operation failed (= error code).

5.2 AllocateWithTag

```
AllocateWithTag(INTEGER:Size
                HIDDEN:Tag);
                INTEGER:Result
                ADDRESS:BlockPointer

XAllocateWithTag(INTEGER:Size
                HIDDEN:Tag);
                ADDRESS:Blockpointer
```

Action

As for Allocate but using given Tag.

Call

Size	As for Allocate.
Tag	Tag to be used.

Return

Result	$>=0$, Operation successful (=bytes allocated). <0, Error (=error code).

5.3 SetStoretag

SetStoreTag(HIDDEN:Tag);
HIDDEN:Oldtag

XSetStoreTag(HIDDEN:Tag);
HIDDEN:Oldtag

Action

Supply a new value for the StoreTag. This value will be used as the tag for all subsequent allocations where a tag is not specified explicitly (i.e. where Allocate is used rather than AllocateWithTag). A library call is provided to deallocate all blocks with the same value of tag.

Call

Tag	New value for the block tag. (This must be obtained using GetNewTag).

Return

Oldtag	The previous value of the tag.

5.4 Deallocate

Deallocate(ADDRESS:BlockPointer);
INTEGER:Result

XDeallocate(ADDRESS:BlockPointer)

Action

Deallocate a previously allocated block of store.

Call

BlockPointer Pointer to the block to be freed.

Return

Result $>=0$, Operation successful.
<0, Operation failed (=error code).

5.5 DeallocateGroup

DeallocateGroup(HIDDEN:Tag);
INTEGER:Result

XDeallocateGroup(HIDDEN:Tag)

Action

Deallocate all blocks allocated with the given tag.

Call

Tag — the Tag of the group to be removed.

Return

Result — $>=0$, Operation successful.
<0, Operation failed (=error code, e.g. invalid Tag).

5.6 GetNewTag

GetNewTag();HIDDEN:Tag

XGetNewTag();HIDDEN:Tag

Action

Return an unused (unique) Store tag.

Call

No parameters.

Return

Tag	The new tag.

5.7 ReturnTag

ReturnTag(HIDDEN:Tag);
INTEGER:Result

XReturnTag(HIDDEN:Tag)

Action

Relinquish the Tag and free any memory allocated with that Tag.

Call

Tag	Tag to be freed.

Return

Result	$>=0$, Operation successful (= Number of memory blocks freed). <0, Operation failed (= error code).

5.8 DeallocateTop

DeallocateTop(BlockAddress: ADDRESS;
SplitAddress: ADDRESS); INTEGER: Result

XDeallocateTop(BlockAddress: ADDRESS
SplitAddress: ADDRESS);

Action

Split an allocated store block into two blocks at the given address and deallocate the block with the higher address.

Call

BlockAddress	Address of the block to be split.
SplitAddress	Address of the section of the block to be deallocated.

Return

Result	$>=0$, Operation successful.
	<0, Operation failed (= error code).

5.9 DeallocateBottom

DeallocateBottom(BlockAddress: ADDRESS;
SplitAddress: ADDRESS); INTEGER: Result;

XDeallocateBottom(BlockAddress: ADDRESS;
SplitAddress: ADDRESS);

Action

Split an allocated store block into two blocks at the given address and deallocate the block with the lower address.

Call

BlockAddress	Address of the block to be split.
SplitAddress	Address of the split point.

Return

Result	$>=0$, Operation successful.
	<0, Operation failed (= error code).

5.10 SetHeapEnd

SetHeapEnd(ADDRESS:HeapEnd);
INTEGER:Result

XSetHeapEnd(ADDRESS:HeapEnd)

Action

Instruct allocator to check future requests for heap against the supplied HeapEnd.

Call

HeapEnd	The new end of heap. Address of first byte not available to be allocated. This address must be > = CurrentHeapEnd.

Return

Result	> = 0, Operation successful. < 0, Operation failed (= error code).

5.11 ResetHeapEnd

ResetHeapEnd();INTEGER:Result

XResetHeapEnd()

Action

Instruct allocator to check that future requests for heap will not result in overlap with the user stack.

Return

Result	$>=0$, Operation successful. <0, Operation failed (=error code).

5.12 CurrentHeapEnd

CurrentHeapEnd:ADDRESS

CurrentHeapEnd contains the address of the first byte above all claimed heap memory. This exported item is for user stack checking and must not be written to.

5.13 GetStoreInformation

GetStoreInformation();Result:Integer
RECORD(Info):StoreInformation

XGetStoreInformation();RECORD(Info):StoreInformation

Action

Get data about current store usage.

Call

No parameters.

Return

Result $>=0$, operation successful,
StoreInformation contains the data.
<0, operation failed (=error code).

The record Info has the following format:

```
RECORD
   ADDRESS  (* base Address of heap       *)
   CARDINAL (* size of heap               *)
   CARDINAL (* total free space           *)
   CARDINAL (* largest free block         *)
   CARDINAL (* total free Module space    *)
   CARDINAL (* largest free Module block  *)
END
```

6 I/O Library

Input/Output operations are carried out by calls to procedures residing in moduleIO . The I/O system is based upon the concept of streams. When a connection is established to an I/O object the I/O library will return a stream number which is used to refer to the object in future transactions. The maximum number of simultaneously open streams is system dependent. The syntax of an I/O object is one of:

(a) <devicename>:
(b) <filing system name>:<filespec>
(c) <filespec>

For case (c) the current filing system is used. The case of letters in all contexts is not significant, other than that the case of the stored name of a file is preserved as that specified in the call which created it. For example FindOutput("Data_3") will create a file called "Data_3" which may be referred to as "Data_3", "DATA_3", "data_3", "daTa_3" etc.

File System names

The various file systems are listed below:

DFS: disc filing system
ADFS: advanced disc filing system
NFS: network filing system

The valid characters which may occur as part of a basic filename (i.e. the 'leaf' name without any directory/drive prefix) are limited to:

A..Z
a..z
0..9
_ and !

A leafname has two components: the base name and the extension, separated by a hyphen "-" (e.g. plot-aof, bench-f77). It is possible (though not recommended) to have files with a null extension. These can be referred

to as e.g. "file3-", though in some contexts the trailing hyphen can be omitted.

The base name can be 1-10 characters.

The extension can be 0-3 characters.

(However in some cases the BBC machine filing systems will impose stronger constraints, for example on DFS the base name is limited to 7 characters.)

Device names

vdu: Refers to the screen (i.e. output only) with filtering of control characters. Only printing ASCII characters (32..126), clear-screen (FF),newline (NL = LF) and carriage-return (CR) are sent to the screen. All others are ignored.

rawvdu: Refers to the screen (for output only).The effect is exactly as defined in the *BBC Microcomputer System User Guide* or equivalent manual.

kb: Refers to the machine's keyboard (input only) with both carriage-return (CR) and line-feed (LF) being read as newline (NL).

Notes: The I/O system maintains an input buffer from which characters are read for kb: requests. When this buffer is empty further requests will cause it to be replenished from the keyboard until a line terminator NL or CR is typed. The characters read are echoed to the vdu: device. Line editing is enabled, i.e.

DELETE	delete character
CTRL - U	delete line
CTRL - D	end of file

rawkb: Refers to the keyboard (input only) with no translation or filtering of characters.

Notes: Raw characters are read directly from the keyboard and are not echoed. If when a request for a raw character is made there are characters in the kb: buffer then they are discarded.

bbc: A combination of rawvdu: for output and rawkb: for input.

tty: A combination of vdu: for output and kb: for input.

rs423: Refers to the serial line (input or output).

printer: (or lp:) Refers to the printer (output only).

null: Refers to a 'sink'. Output to this device is discarded. On input it always returns EndOfFile.

Special Devices

Input: Refers to the Current Input Stream so FindInput("Input:") is equivalent to InputStream().

Output: Refers to the Current Output Stream so FindOutput("Output:") is equivalent to OutputStream().

Control: Refers to the Curent Control Stream so FindInput("Control:") is equivalent to ControlStream().

Error: Refers to the Current Error Stream so FindOutput("Error:") is equivalent to ErrorStream().

6.1 FindInput

FindInput(STRING:Destination);
INTEGER:Result

XFindInput(STRING:Destination);
CARDINAL:StreamNumber

Action

Connect a stream to the named I/O object for input.

Call

Destination a string specifying an I/O object.

Return

Result $>= 0$, Operation successful (= stream number).
< 0, Operation failed (=error code).

6.2 FindOutput

FindOutput(STRING:Destination);
INTEGER:Result

XFindOutput(STRING:Destination);
CARDINAL:StreamNumber

Action

Connect a stream to the named I/O object for output.

Call

Destination a string describing an I/O object.

Return

Result $>= 0$, Operation successful (= stream number).
< 0, Operation failed (= error code).

6.3 FindUpdate

FindUpdate(STRING: Destination);
INTEGER:Result

XFindUpdate(STRING: Destination);
CARDINAL:StreamNumber

Action

Connect a stream to the named I/O object for input/output.

Call

Destination a string describing an I/O object.

Return

Result $>= 0$, Operation successful (= stream number).
< 0, Operation failed (= error code).

6.4 CloseStream

CloseStream(CARDINAL:Stream);
INTEGER:Result

XCloseStream(CARDINAL:Stream)

Action

Close a stream.

Call

Stream The number of the stream to be closed.

Return

Result $>= 0$, Operation successful.
< 0, Operation failed(= error code) e.g. stream not open.

6.5 SelectInput

SelectInput(CARDINAL:Stream);
INTEGER:Result

XSelectInput(CARDINAL:Stream)

Action

Select the stream to be used for input using the I/O procedure calls which do not take a stream parameter (e.g. ReadByte). The selected input stream is preserved over program invocation.

Call

Stream The number of the stream to be used.

Return

Result $>=0$, Operation successful.
<0, Operation failed (= error code).

6.6 SelectOutput

SelectOutput(CARDINAL:Stream);
INTEGER:Result

XSelectOutput(CARDINAL:Stream)

Action

Select the stream to be used for output using the I/O procedure calls which do not take a stream parameter (e.g. XBlockWrite). The selected output stream is preserved over program invocation.

Call

Stream The number of the stream to be used.

Return

Result $>=0$, Operation successful.
<0, Operation failed (=error code).

6.7 SelectUpdate

SelectUpdate(CARDINAL:Stream);
INTEGER:Result

XSelectUpdate(CARDINAL:Stream)

Action

Select the stream to be used for both input and output. Equivalent to SelectOutput(Stream) then SelectInput(stream).

Call

Stream — The number of the stream to be used (returned from FindUpdate).

Return

Result — $>=0$, Operation successful.
<0, Operation failed (=error code).

6.8 SetErrorStream

SetErrorStream(CARDINAL:Stream);
INTEGER:Result

XSetErrorStream(CARDINAL:Stream)

Action

Set the stream to be used for error output (i.e. that returned by ErrorStream()). The error stream is preserved across program invocation.

Call

Stream the stream to be the new error stream.

Return

Result $>= 0$, Operation successful.
< 0, Operation failed = error code.

6.9 SetControlStream

SetControlStream(CARDINAL:Stream);
INTEGER:Result

XSetControlStream(CARDINAL:Stream)

Action

set the stream to be used for control input, i.e. that returned by ControlStream(). The control stream is preserved across program invocation.

Call

Stream The stream to be the control stream.

Return

Result $>=0$, Operation successful.
<0, Operation failed (=error code).

6.10 InputStream

InputStream();INTEGER:Result

XInputStream();CARDINAL:Result

Action

Return the stream number of the currently selected input stream.

Call

No parameters.

Return

Result	$>=0$, Stream number of current input stream. <0, Operation failed (= error code).

6.11 OutputStream

OutputStream();INTEGER:Result

XOutputStream();CARDINAL:Result

Action

Return the stream number of the currently selected output stream.

Call

No parameters.

Return

Result $>=0$, Stream number of current output stream.
<0, Operation failed (=error code).

6.12 ErrorStream

ErrorStream();INTEGER:Result

XErrorStream();CARDINAL:Result

Action

Return the stream number of the currently selected error stream.

Call

No parameters

Return

Result $>=0$, Stream number of current error stream.
<0, operation failed (=error code).

6.13 ControlStream

ControlStream();INTEGER:Result

XControlStream();CARDINAL:Result

Action

Return the stream number of the currently selected control stream.

Call

No parameters.

Return

Result $>=0$, Stream number of current control stream.
<0, Operation failed (=error code).

6.14 WriteByte

WriteByte(CARDINAL:Byte);
INTEGER:Result

XWriteByte(CARDINAL:Byte)

Action

Write a byte to the currently selected output stream.

Call

Byte the byte to be written.

Return

Result $>=0$, Operation successful.
<0, Operation failed (=error code).

6.15 ReadByte

ReadByte();INTEGER: Result

XReadByte();CARDINAL:Byte

Action

Read a byte from the currently selected input stream.

Call

No parameters.

Return

Result $>=0$, Operation successful (= zero extended byte read).
<0, Operation failed (=error code).

6.16 CurrentByte

CurrentByte();INTEGER:Result

XCurrentByte();INTEGER:Result

Action

Return the byte which will be supplied by the next read on the currently selected input stream.

Call

No parameters.

Return

Result $>=0$, Operation successful (= byte read).
<0, Operation failed(= error code).

6.17 BlockRead

```
BlockRead(CARDINAL:Blength
          ADDRESS:Buffer);
          INTEGER:Result
          CARDINAL:BytesRead

XBlockRead(CARDINAL:Blength
           ADDRESS:Buffer);
           CARDINAL:BytesRead
```

Action

Read a block of bytes from the currently selected input stream. The number of bytes supplied will be the minimum of Blength and the number of bytes left before end of file.

Notes: This operation may be used on a stream connected to the keyboard, in which case the number of bytes read will be determined by the number of characters typed up to, and including, the terminating character (NL on kb: and tt:, CR on rawkb: and bbc:).

Call

Blength	Maximum length to read into buffer.
Buffer	Address of buffer for the data.

Return

Result	$>=0$, Operation successful (=number of bytes placed in the buffer). <0, Operation failed (=error code).
BytesRead	Number of bytes placed in buffer.

6.18 BlockWrite

```
BlockWrite(CARDINAL:Blength
           ADDRESS:Buffer);
           INTEGER:Result
           CARDINAL:BytesWritten

XBlockWrite(CARDINAL:Blength
           ADDRESS:Buffer);
```

Action

Write a block of bytes to the currently selected output stream.

Call

Blength	Number of bytes to write.
Buffer	Address of buffer containing the bytes.

Return

Result	>=0, Operation successful. <0, Operation failed (=error code).
BytesWritten	Number of bytes written.

6.19 SWriteByte

SWriteByte(CARDINAL:Stream
CARDINAL:Byte);
INTEGER:Result

XSWriteByte(CARDINAL:Stream
CARDINAL:Byte)

Action

As WriteByte, but use 'stream' rather than currently selected default output stream.

6.20 SReadByte

SReadByte(CARDINAL:Stream);
INTEGER:Result

XSReadByte(CARDINAL:Stream);
CARDINAL:Byte

Action

As ReadByte, but use 'stream' rather than currently selected default input stream.

6.21 SCurrentByte

SCurrentByte(CARDINAL:Stream);
INTEGER:Result

XSCurrentByte(CARDINAL:Stream);
INTEGER:Result

Action

As for CurrentByte, but use 'stream' rather than currently selected default input stream.

6.22 SBlockRead

```
SBlockRead(CARDINAL:Stream
           CARDINAL:Blength
           ADDRESS:Buffer);
           INTEGER:Result
           CARDINAL: BytesRead

XSBlockRead(CARDINAL:Stream
           CARDINAL:Blength
           ADDRESS:Buffer);
           CARDINAL: BytesRead
```

Action

As for BlockRead, but use 'stream' rather than currently selected default input stream.

6.23 SBlockWrite

```
SBlockWrite(CARDINAL:Stream
            CARDINAL:Blength
            ADDRESS:Buffer);
            INTEGER:Result
            CARDINAL:BytesWritten

XSBlockWrite(CARDINAL:Stream
             CARDINAL:Blength
             ADDRESS:Buffer);
```

Action

As for BlockWrite, but use 'stream' rather than currently selected default output stream.

6.24 GetFileOffset

GetFileOffset(CARDINAL:Stream);
INTEGER:Result
CARDINAL:Offset

XGetFileOffset(CARDINAL:Stream);
CARDINAL:Offset

Action

Return the position of the file pointer for the file associated with the given stream. The returned value will be 0 if no bytes have been read or written on the stream.

Call

Stream A stream.

Return

Result $>=0$, Operation successful and second result is file pointer).
<0, Operation failed(= error code). It will fail if stream is not a file stream (e.g. kb:,vdu: etc).

6.25 SetFileOffset

SetFileOffset(CARDINAL:Stream
CARDINAL:Offset);
INTEGER:Result

XSetFileOffset(CARDINAL:Stream
CARDINAL:Offset)

Action

Set the file pointer for the file associated with the given stream.

Call

Stream	A stream.
Offset	New value of the file pointer.

Return

Result	$>=0$, Operation successful. <0, Operation failed (= error code).

6.26 BytesOutstanding

BytesOutstanding(CARDINAL:Stream);
INTEGER:Result
CARDINAL:Available

XBytesOutstanding(CARDINAL:Stream);
CARDINAL:Available

Action

Return the number of bytes available immediately on the given stream. For disc files this will be the number of bytes left in the file. For a terminal stream it will be the number of characters in the input buffer.

Call

Stream	Stream number

Return

Result	$>=0$, Operation successful (Available = number of bytes available). <0, Operation failed (=error code).

6.27 EndOfFile

EndOfFile(CARDINAL:Stream);
INTEGER:Result

XEndOfFile(CARDINAL:Stream);
BOOLEAN:Result

Action

Inform caller if specified input stream is at end of file.

Call

Stream The stream.

Return

(EndOfFile)

Result
< 0 Error, stream is invalid.
= 0 False, not at end of file.
= 1 True, at end of file.

or (XEndOfFile)

Result
= False, not end of file.
= True, is end of file.

6.28 FlushOutput

FlushOutput();INTEGER:Result

XFlushOutput();

Action

Flushes current output stream. A FlushOutput is done implicitly when CloseStream is called on an output stream.

Call

No parameters.

Return

Result	$>=0$, Operation successful.
	<0, Operation failed (=error code).

6.29 SFlushOutput

SFlushOutput(CARDINAL:Stream);
INTEGER:Result

XSFlushOutput(CARDINAL:Stream);

Action

Flushes The specified output stream. A FlushOutput is done implicitly when CloseStream is called on an output stream.

Call

Stream The stream to be flushed.

Return

Result $>=0$, Operation successful.
<0, Operation failed (=error code).

6.30 DeviceType

DeviceType(STRING:Name); Result: INTEGER
RECORD (IOCapabilities): Capabilities

XDeviceType(STRING:Name);RECORD (IOCapabilities): IOCapabilities

Action

Return information about the device representated by the given name.

Call

Name — IO device name and data (if applicable). Format as for FindInput etc.

Return

Result — $>=0$, Operation successful. The Capabilities record contains information as shown below.
<0, Operation failed (= error code).

The format of the capabilities record is:

```
RECORD
   DOUBLE   (* Set of operations *)
   CARDINAL (* Device Width *)
   CARDINAL (* Device height *)
END
```

The operations bits are:

Bit	Meaning if bit is SET
0	Device is media
1	Device is interactive
2	Device will accept FindInput
3	Device will accept FindOutput
4	Device will accept FindUpdate
5	Device will accept Seek operations
6	Device output is filtered

Device width and Device height are set to 0 if the device has no notion of these concepts.

6.31 StreamType

StreamType(CARDINAL: Stream); Result: INTEGER
RECORD (IOCapabilities): Capabilities

XDeviceType(CARDINAL: Stream);RECORD (IOCapabilities):
IOCapabilities

Action

Return information about the device associated with the given stream.

Call

Stream Stream number of an open stream.

Return

Result $>=0$, Operation successful. The Capabilities record contains information about the capabilities of the device attached to the stream. (Format as capability record in 'DeviceType').
<0, Operation failed (= error code).

6.32 SetTabs

SetTabs(STRING: TabStops); INTEGER: Result

XSetTabs(STRING: TabStops);

Action

Set the system tab string. This is the information used to control the expansion of tabs on output devices.

Call

TabStops	A string (maximum of 256 chars) having the character '*' in those column positions which are to be tab stops.

Return

Result	$>=0$, Operation successful. <0, Operation failed (= error code).

6.33 GetTabs

GetTabs(); INTEGER: Result
STRING: TabStops;

XGetTabs(); STRING: TabStops;

Action

Get the system tab control string. (The maximum length of this string is 256 characters, the returned string will be truncated to the length of the callers buffer.)

Call

No parameters.

Return

Result $>=0$, Operation successful. TabStops will be space filled and have '*' characters placed at the tab positions.
<0, Operation failed (= error code).

7 File Support

Procedures are provided in moduleFile to carry out filing system operations. The PANOS filing system is built on top of the BBC machine filing system. All file names presented to PANOS are translated to convert them into their BBC machine form. The BBC file name is called the 'Physical file name'. The translation is carried out in two parts. Firstly, if the file name is a relative one (i.e. its path does not begin with '&' or '$') then the working directory for its filing system is prepended. Secondly the leafname is translated using the 'File$' global variables for the substitution of filename extensions. See the *Panos Guide to Operations* for full details.

7.1 GetDateStamp

GetDateStamp(STRING:FileName);
INTEGER:Result
RECORD(Btim):DateStamp

XGetDateStamp(STRING:FileName);
RECORD(Btim):DateStamp

Action

Returns the Binary time stamped on the named file, if present.

Call

Filename Name of file.

Return

Result $>=0$, File has a valid datestamp, returned in DateStamp.
<0, Error (=error code).

7.2 SetDateStamp

```
SetDateStamp(STRING:FileName
             RECORD(BTim):DateStamp);
             INTEGER:Result

XSetDateStamp(STRING:Filename
              RECORD(BTim):DateStamp)
```

Action

Set the datestamp of the named file to the given value.

Call

Filename Name of the file.

DateStamp New value for files datestamp.

The BTim format is defined under the description of module TimeAndDate.

Return

Result $>=0$, Operation successful.
<0, Error (=error code).

7.3 Touch

Touch(STRING:FileName);
INTEGER:Result

XTouch(STRING:FileName)

Action

If the given file has a valid datestamp then it will be updated to be stamped with the current time.

Call

FileName Name of file.

Return

Result >=0 File DateStamp has been updated.
<0 Error (=Error Code).

7.4 RenameFile

```
RenameFile(STRING:OldName
           STRING:NewName);
           INTEGER:Result

XRenameFile(STRING:OldName
            STRING:NewName)
```

Action

Rename a file.

Note: It is not possible to rename a file across filing systems, or drives.

Call

OldName	File to be renamed.
NewName	New name for the file.

Return

Result	$>=0$, File renamed.
	<0, Error (=errorcode).

7.5 DeleteFile

DeleteFile(STRING:Filename);
INTEGER:Result

XDeleteFile(STRING:Filename)

Action

Delete a file.

Call

Filename	File to be deleted.

Return

Result	$>=0$, Operation succeeded. <0, Error (=error code).

7.6 PhysicalFileName

PhysicalFileName(STRING:Filename);
INTEGER:Result
STRING:FilingSystem
STRING:PhysicalFileName

XPhysicalFileName(STRING:Filename);
STRING:FilingSystem
STRING:PhysicalFileName

Action

Transform the Panos filename into its BBC filing system form according to the current environment (i.e. the current working directory and any file$-xxx variables).

Call

Filename — A Panos filename.

Return

Result — $>=0$, Operation Successful.
<0, Operation failed (=error code).

FilingSystem
Textual name of filing system in which the file resides.

PhysicalFileName
Expanded file name (derived from Panos file name transformed by extensions and working directory).

7.7 SetWorkingDirectory

SetWorkingDirectory(STRING:Path);
INTEGER:Result

XSetWorkingDirectory(STRING:Path)

Action

Informs the Panos file manager of the position of the working directory. This call must be used instead of the BBC *DIR command. Its effect is similar. If the path contains a filing system name then this becomes the current filing system. If the new filing system is different from the previous one then the working directory in the old filing system is not changed and may be referred to by '<filing system name>:', e.g. nfs:

Call

Path	Panos file name (of a directory).

Return

Result	>=0, Operation succesful. <0, Error (=error code); working directory has not been changed.

7.8 GetWorkingDirectory

GetWorkingDirectory();INTEGER: Result
STRING: Path

XGetWorkingDirectory(); STRING: Path

Action

Return the current working directory.

Call

No parameters.

Return

Result	$>=0$, Operation successful. Path contains the path (including filing system name) to the current working directory. <0, Operation failed (= error code).

7.9 LoadFile

```
LoadFile(STRING: FileName
         CARDINAL: BufferSize
         ADDRESS: Buffer); INTEGER: Result

XLoadFile(STRING: FileName
          CARDINAL: BufferSize
          ADDRESS: Buffer);
```

Action

Loads the whole of the given file into the callers buffer.

Call

FileName	The name of the file to be loaded.
BufferSize	The size of the callers buffer.
Buffer	The address of the start of the buffer.

Return

Result	$>=0$, Operation successful. <0 , Operation failed (= error code).

7.10 SaveFile

```
SaveFile(STRING: FileName
         CARDINAL: BufferSize
         ADDRESS: Buffer); INTEGER:Result

XSaveFile(STRING: FileName
          CARDINAL: BufferSize
          ADDRESS: Buffer);
```

Action

Save a buffer to a file in one operation.

Call

FileName	The name of the file to hold the data.
BufferSize	The number of bytes to be saved.
Buffer	The address of the first byte to be saved.

Return

Result	$>=0$, Operation successful. <0 , Operation failed (= error code).

7.11 PhysicalDirRead

PhysicalDirRead(STRING: DirName
PROCEDURE: UserProcedure
HIDDEN: Argument);
INTEGER: Result

XPhysicalDirRead(STRING: DirName
PROCEDURE: UserProcedure
HIDDEN: Argument); CARDINAL: Result

Action

For each entry in the given directory call the user procedure with the supplied argument. The procedure must be of form

UserProcedure(STRING: FileName
HIDDEN: Argument);INTEGER: Result

If any of these calls to the user procedure returns a negative result then PhysicalDirRead returns immediately with that result.

Call

Dirname	the directory to be read.
UserProcedure	Procedure to be called.
Argument	Argument to be given to the user procedure.

Return

Result	$>=0$, Operation successful (= number of calls made). <0 , Operation failed (= error code).

7.12 InitDirRead

InitDirRead(STRING: DirName
BOOLEAN: ReadInfo
BOOLEAN: TargetIsDir
STRING: Target); INTEGER: Result
HIDDEN: Handle

XInitDirRead(STRING: DirName
BOOLEAN: ReadInfo
BOOLEAN: TargetIsDir
STRING: Target); HIDDEN: Handle

Action

Reads content of a PANOS directory, returning a handle to use to request these names.

Call

If ReadInfo is FALSE, then only the name and length fields of returned infoRec's are valid (and a slight gain in speed is achieved).

If TargetIsDir, then entries matching extension templates (other than the 'identity' template '-') are ignored, resulting in a gain of speed.

If Target is non-null and has a non-wild extension part, then (as an optimization) only the directory into which that extension maps will be read.

Return

Result $>=0$, Operation successful. Handle is identifier for this directory. Result is the number of entries in the directory.
<0, Operation failed (= error code).

7.13 GetDirEntry

```
GetDirEntry (CARDINAL: Index
                        HIDDEN: Handle); INTEGER: Result
                            STRING: FileName
                            RECORD: (InfoRec) Information

XGetDirEntry (CARDINAL: Index
                        HIDDEN: Handle); STRING: FileName;
                            RECORD (InfoRec): Information
```

Action

Returns information about a directory entry (see InitDirRead).

Call

Index	The index of the entry required.
Handle	As returned by InitDirRead.

Return

Result	$>=0$, Operation successful. (Filename and information for this instance are set up.) <0 , Operation failed (= error code).

The format of InfoRec is:

```
            RECORD
            BOOLEAN          (* Is directory *)
            RECORD (FileData) (* File Information *)
            RECORD (BTim)     (* TimeStamp *)
  END
```

Note: *see* GetFileInformation for format of FileData record, and Module TimeAndDate for the BTim record.

7.14 EndDirRead

EndDirRead (HIDDEN: Handle); INTEGER: Result

XEndDirRead (HIDDEN: Handle);

Action

Terminate processing for a given directory.

Call

Handle	Identifier returned by InitDirRead.

Return

Result	$>=0$, Operation successful. <0 , Operation failed (= error code).

7.15 IsWild

IsWild(STRING: FileName); BOOLEAN: Result;

XIsWild(STRING: FileName); BOOLEAN: Result;

Action

Determine whether a file name contains wild characters.

Call

Filename	The filename to be tested.

Return

Result	TRUE, the FileName is wild. FALSE, the FileName is not wild.

7.16 FileReplace

FileReplace (STRING: wildA
STRING: instanceA
STRING: wildB);
INTEGER: Result
STRING: CorrInstanceB

XFileReplace (STRING: wildA
STRING: instanceA
STRING: wildB);
STRING: CorrInstanceB

Action

Creates CorrInstanceB as being WildB with the wild specifiers being substituted for the instantiations of the corresponding wild specifiers in WildA needed to match against InstanceA.

Call

WildA and WildB are wild filenames with the same sequence of wild specifiers. InstanceA is a fully specified filename matching WildA.

Example:
FileReplace("*-mod", "Fred-mod", "*-mud") yields "Fred-mud".

Return

Result $>=0$, Operation successful - CorrInstanceB set up as specified.
<0, Operation failed (=error code).

7.17 Expand

Expand (STRING: WildName
PROCEDURE: ProcessProc
HIDDEN: ProcessArg
BOOLEAN: TargetIsDir);INTEGER: Result

XExpand (STRING: WildName
PROCEDURE: ProcessProc
HIDDEN: ProcessArg
BOOLEAN: TargetIsDir);

Action

For each match against WildName, Expand makes a call:

ProcessProc (Instance, ProcessArg),

where Instance is an instantiation of WildName.

If any ProcessProc call returns an error result (i.e. < 0), then expand returns immediately with that as its result.

WildName is a wild PANOS pathname, with following wild specifiers:

? Matches any single character.
* Matches any zero or more characters, within a name.
... Matches any zero or more names (i.e. a section of a pathname).

Return

Result $>= 0$, Success (= number of ProcessProc calls made).
< 0, Error (= error code).

Examples:

Expand ("*-mod", PProc, PArg) generates calls:

PProc ("Fred-mod", PArg)
PProc ("Jim-mod", PArg)
(.. etc ..)

Expand ("ADFS:...c??t", PProc, PArg) generates calls:

PProc ("ADFS:coot", PArg)
PProc ("ADFS:A.B.coat", PArg)
(.. etc ..)

Limitations:

i. Filing system and drive prefixes (if any) cannot be wild.
ii. At most one "..." specifier in a pathname.

7.18 GetFileInformation

GetFileInformation(STRING: FileName);INTEGER: Result
RECORD(FileData) CatalogueInfo;
RECORD (BTim) DateStamp;

XGetFileInformation(STRING: FileName);
RECORD(FileData) CatalogueInfo;
RECORD (BTim) DateStamp;

Action

Return the catalogue information for a given file.

Call

FileName The file for which information is required.

Return

Result $>=0$, Operation successful.
(= File type:
$=1$ -> File
$=2$ -> directory)
CatalogueInfo and DateStamp are set up.

<0 , Operation failed (= error code).

The format of RECORD (BTim) is given in 'Time and Date' (q.v.) Note that if the file does not have a valid datestamp the record will have all its fields returned as zero.

The format of RECORD (FileData) is:

```
RECORD
  CARDINAL (* File load address *)
  CARDINAL (* File execution address *)
  CARDINAL (* File length *)
  CARDINAL (* BBC filing system attributes *)
END
```

7.19 SetFileInformation

```
SetFileInformation(STRING: FileName
                   RECORD(FileData) CatalogueInfo;
                   RECORD (BTim) DateStamp);
                   INTEGER: Result

XSetFileInformation(STRING: FileName
                   RECORD(FileData) CatalogueInfo;
                   RECORD (BTim) DateStamp);
```

Action

Set the catalogue information for a given file.

Call

FileName	The file for which information is required.
CatalogueInfo	The files catalog information (see 'GetFileInformation').
DateStamp	The required datestamp.

Return

Result	$>=0$, Operation successful. <0 , Operation failed (= error code).

7.20 CreateFile

CreateFile(STRING: FileName
CARDINAL: Size); INTEGER: Result

XCreateFile(STRING: FileName
CARDINAL: Size);

Action

Create a file.

Call

FileName	The name of the file to be created.
Size	The size (in bytes) of the file to be created.

Return

Result	$>=0$, Operation successful. <0, Operation failed (= error code).

7.21 CreateDirectory

CreateDirectory(STRING: Name); INTEGER: Result

XCreateDirectory(STRING: Name);

Action

Create a directory.

Call

Name — The name of the directory to be created.

Return

Result — $>=0$, Operation successful.
<0, Operation failed (= error code).

8 Loader

The Loader is the part of Panos responsible for the loading of user's program images. When a program is loaded the loader resolves references from it to Panos services. It is possible for user provided procedures and data objects to be added dynamically to the Panos system. Two Panos procedures are available to permit this they reside in the moduleLoader and they are detailed in the following pages.

8.1 DeclareProc

DeclareProc(STRING: ModuleName
STRING: ProcedureName
PROCEDURE: normalform
PROCEDURE: eventform);INTEGER:ReturnCode

XDeclareProc(STRING: ModuleName
STRING: ProcedureName
PROCEDURE: normalform
PROCEDURE: eventform);

Action

Declare a procedure to the Panos loader. This procedure will be visible to all program environments loaded (e.g. by Program.Run) following this call, until the program environment which made the call is itself terminated.

Call

ModuleName	Name of the module in which this procedure resides.
ProcedureName	Name of the procedure being declared (loader will automatically generate the 'X' variant name.
normalform	Descriptor for non event form of the procedure.
eventform	Descriptor for the event form of the procedure

Return

ReturnCode	$>=0$,Operation successful.
	<0, Operation failed.

8.2 DeclareData

DeclareData(STRING: ModuleName
STRING: DataName
ADDRESS: item);INTEGER:ReturnCode

XDeclareData(STRING: ModuleName
STRING: DataName
ADDRESS: item);

Action

Declare the address of an external data item to the Panos loader. The same rules apply to the data item as for a user-defined procedure (see DeclareProc).

Call

ModuleName	Name of the module in which the data item resides.
DataName	Name of the item being declared.
item	The address of the declared item.

Return

ReturnCode	>=0,Operation successful.
	<0, Operation failed.

9 Random numbers

Panos supports the generation of pseudo-random numbers. The user may obtain a different series of pseudo-random numbers by re-defining the Seed. The procedures, which reside in the moduleRandom , are described below.

9.1 Random

Random();CARDINAL:RandomNumber

XRandom();CARDINAL:RandomNumber

Action

Return a pseudo-random number.

Call

No parameters.

Return

RandomNumber Next number from the pseudo-random sequence.

9.2 SetRandomSeed

SetRandomSeed(CARDINAL:Seed)

XSetRandomSeed(CARDINAL:Seed)

Action

Set the seed for the generation of the pseudo-random sequence.

Call

Seed The new seed.

Return

No value returned.

10. Time and date

Three representations for the date and time are available. The primitive representation is Binary and two string formats are supported. These are Textual format and Standard format. Examples of the two formats are:

Textual: "16 May 84 20:54:19"

Standard: "1985-02-17 09:57:15.21"

In the following definitions the record BTim is composed of two CARDINAL values:

```
FORMAT BTim (CARDINAL:Low,CARDINAL:High)
```

The two fields together comprise a 64 bit representation of absolute time, measured in centi-seconds from a base point of "1900-01-01 00:00:00.00". The conversion procedures which reside in the moduleTimeAndDate are described in the following sections.

10.1 BinaryTime

BinaryTime();INTEGER:Result
RECORD(BTim):BinaryTime

XBinaryTime();RECORD(BTim):BinaryTime

Action

Read Binary Time.

Call

No parameters.

Return

Result $>=0$, Operation successful and BinaryTime is the current time in system binary format.
<0, Operation failed (=error code).

10.2 SetBinaryTime

SetBinaryTime(RECORD(BTim):BinaryTime);
INTEGER:Result

XSetBinaryTime(RECORD(BTim):BinaryTime)

Action

Set Binary Time.

Call

BinaryTime — The time to be set, in system binary time format.

Return

Result — $>=0$, Operation successful.
<0, Operation failed (=error code).

10.3 BinaryTimeOfStandardTime

BinaryTimeOfStandardTime(STRING:STime);
INTEGER:Result
RECORD(BTim):BinaryTime

XBinaryTimeOfStandardTime(STRING:STime);
RECORD(BTim):BinaryTime

Action

Calculates a BinaryTime record corresponding to the Standard format string.

Call

STime A string in Standard time format.

Return

Result	>=0, Operation successful. <0, Operation failed(=error code).
BinaryTime	The system binary time representation of the parameter.

10.4 BinaryTimeOfTextualTime

BinaryTimeOfTextualTime(STRING:TTime);
INTEGER:Result
RECORD(BTim):BinaryTime

XBinaryTimeOfTextualTime(STRING:TTime);
RECORD(BTim):BinaryTime

Action

Calculates a BinaryTime corresponding to the string in Textual format.

Call

TTime A string in Textual time format.

Return

Result	$>=0$, Operation successful. <0, Operation failed (=error code).
BinaryTime	The system binary time representation of the parameter.

10.5 StandardTimeOfBinaryTime

```
StandardTimeOfBinaryTime(RECORD(BTim) BinaryTime);
        INTEGER: Result
        STRING: STime

XStandardTimeOfBinaryTime(RECORD(BTim) BinaryTime);
        STRING STime
```

Action

Returns a string in the Standard format corresponding to the binary time parameter.

Call

BinaryTime	A system binary time value.

Return

Result	$>=0$, Operation successful. <0, Operation failed (= error code).
STime	Standard time string corresponding to the supplied binary time.

10.6 TextualTimeOfBinaryTime

```
TextualTimeOfBinaryTime(RECORD(BTim):BinaryTime);
        INTEGER:Result
        STRING:TTime

XTextualTimeOfBinaryTime(RECORD(BTim):BinaryTime);
        STRING:TTime
```

Action

Returns a string in the Textual format corresponding to a binary time record.

Call

BinaryTime	A system binary time value.

Return

Result	$>=0$, Operation successful. <0, Operation failed (=error code).
TTime	Textual time version of binary representation.

10.7 Time

Time();INTEGER:Result
STRING:TheTime

XTime();STRING:TheTime

Action

Returns a string giving the current time of day as the time portion of the Textual time format, e.g. "18:07:20".

Call

No parameters.

Return

Result	> =0, Operation successful. <0, Operation failed (=error code).
TheTime	A string containing the time of day.

10.8 StandardTime

StandardTime();INTEGER:Result
STRING:TheTime

XStandardTime();STRING:TheTime

Action

Returns a string in the Standard format giving the current date and time.

Call

No parameters.

Return

Result	$>=0$, Operation successful. <0, Operation failed(= error code).
TheTime	The current date and time in standard format.

10.9 Date

Date();INTEGER: Result
STRING: TheDate

XDate();STRING TheDate

Action

Returns the current date in the same style as is used in the Textual format, e.g. '18 Mar 54'.

Call

No parameters.

Return

Result	$>=0$, Operation successful. <0, Operation failed(= error code).
TheDate	The date in (partial) Textual format.

10.10 TimeAndDate

TimeAndDate();INTEGER:Result
STRING:TheTimeAndDate

XTimeAndDate();STRING:TheTimeAndDate

Action

Returns the current time and date as a string in the Textual format.

Call

No parameters.

Return

Result $>=0$, Operation successful.
<0, Operation failed (=error code).

TheTimeAndDate is the current date and time in (partial) Textual format.

11. Condition Handlers

This section is primarily of interest to language implementors. It describes the condition handler procedure which may be included in a compiled module at compile (or assembly) time. The handler is called at various times by the operating system, principally when an exception occurs, but also when the module is initialised and when it terminates.

The major part of this section is taken up with descriptions of when the operating system calls the handler. In addition, the procedure Signal is described. This allows users to call the condition handler, and may therefore be used to provide a 'soft error' facility. The procedures provided to control the condition handler reside in moduleHandler .

Exceptions

Exceptions are synchronous to the flow of program execution. They are caused directly by the execution of the code in a program. For example, exceptions include:

(1) the 32016 Hardware exceptions, e.g. illegal instruction, breakpoint trap etc.

(2) 'soft' exceptions generated by calls to the Signal procedure.

If an exception occurs in a module then its Condition handler is entered with the parameter CallType = Exception. If the handler is unable to handle the exception then Panos can pass on the exception to other handlers (with CallType = ExceptionPassedOn) until a successful recovery has been made.

If any handler is able to recover from the exception condition then Panos resumes execution of the program.

The behaviour of Panos on exceptions is described by the following pseudo-code. Note that all exception processing is performed on a small stack separate from the ordinary user stack. As a consequence user exception handlers should be written so as to use minimal stack (no more than 1 Kb), and be careful not to generate an exception while they execute (since this is fatal to the program).

```
TYPE HandlerFn =
  FUNCTION (INTEGER, INTEGER, RECORD(Environment)
       REF * 2): INTEGER

  FUNCTION  ModuleHandler (RECORD(Environment):
                    Env): HandlerFn
    BEGIN
      RETURN ...    ! Returns handler function for module indicated
              ! by MOD field of Environment parameter.
    END FUNCTION

  FUNCTION  HasBeenDescribed (INTEGER: status): BOOLEAN
```

```
  BEGIN
    !
    ! Check the Info field of an error status for
    ! 0, which implies that a message about this
    ! error has already been printed.
    !
    RETURN status<:28,3> = 0
  END FUNCTION

PROCEDURE MarkAsHavingBeenDescribed (VAR INTEGER: status)
  BEGIN
    !
    ! Clear the Info field of an error status
    !
    status<:28,3> := 0
  END PROCEDURE

PROCEDURE DiagnoseException (INTEGER: Code,
                    RECORD(Environment): ExceptionEnv)
  PROCEDURE Diagnose (RECORD(Environment): Env)
    BEGIN
      !
      ! Prints error message corresponding to Code, in
      ! standard format on error stream.  For hardware
      ! exceptions some details of the environment are
      ! also displayed.
      !
      ...
    END PROCEDURE

  RECORD(Environment): CurrentEnv
  HandlerFn:  Handler
  INTEGER: status

  BEGIN
    CurrentEnv := ExceptionEnv   ! start from the exception point
    Handler := ModuleHandler (CurrentEnv)
    status := Handler (Diagnose, Code, CurrentEnv, ExceptionEnv)
    LOOP
      EXIT IF status = 1        ! diagnosis of error produced
```

```
      status := Handler (Unwind, 0, CurrentEnv, NIL)
      EXIT IF status # 1
      Handler := ModuleHandler (CurrentEnv)
      status := Handler (DiagnosePassedOn, Code,
                  CurrentEnv, ExceptionEnv)
    END LOOP
    IF status = 1 THEN
      !
      ! Diagnosis has been output - nothing to do.
      !
    ELSE
      Diagnose (Code)
    END IF
  END PROCEDURE

PROCEDURE DescribeEnvironment (RECORD(Environment): Env)

  PROCEDURE ClearModuleDataFlags
    BEGIN
      !
      ! Using information about the current loaded image,
      ! a private flag corresponding to each loaded module
      ! is cleared down, indicating that no diagnostics
      ! about that module's static data have been given.
      !
      ...
    END PROCEDURE

  FUNCTION ModuleDataDescribed (RECORD(Environment): Env):
BOOLEAN REF
    BEGIN
      RETURN ...   ! returns ref to private flag specific to active
              ! module in Env.
    END FUNCTION

  PROCEDURE GiveMinimalFrameDescription (INTEGER: Level,
                          RECORD(Environment): Env)
    BEGIN
      !
      ! Prints a single line showing PC and module (named if possible)
      ! of given environment.  If level = 0 then this is prefixed
```

```
    ! with "Executing at", otherwise with "called from".
    !
    ...
  END PROCEDURE

RECORD(Environment): CurrentEnv
HandlerFn: Handler
BOOLEAN REF: StaticDataDescribed
INTEGER: Status, Level

BEGIN   ! DescribeEnvironment
  !
  ! Scan through the handlers asking each to show the state of
  ! the frame at the respective level. All modules are initially
  ! marked as not having had description of their module (static)
  ! data given; at each level, if the current module is so marked,
  ! a DiagnoseModuleData call is also performed, and the mark is
  ! cleared.
  !
  CurrentEnv := ExceptionEnv; Level := 0
  ClearModuleDataFlags
  REPEAT
    Handler := ModuleHandler (CurrentEnv)
    Status := Handler (DescribeFrame, Level, CurrentEnv, NIL)
    IF Status = 1 THEN
      !
      ! Frame described by user's handler.
      !
    ELSE
      GiveMinimalFrameDescription (CurrentEnv)
    END IF
    StaticDataDescribed := ModuleDataDescribed (CurrentEnv)
    IF StaticDataDescribed THEN
      !
      ! Already done.
      !
    ELSE
      Status := Handler (DescribeModuleData, 0, CurrentEnv, NIL)
      !
      ! Ignore result status - nothing can be done...
      !
```

```
        StaticDataDescribed := TRUE
      END IF
      Status := Handler (Unwind, 0, CurrentEnv, NIL)
      Level := Level + 1
    UNTIL Status # 1
  END PROCEDURE

PROCEDURE  Abandon (INTEGER: code)
  BEGIN
    Program.Stop (Code)
  END PROCEDURE

STATIC BOOLEAN: ProcessingException := FALSE

PROCEDURE  ResumeEnvironment (RECORD(Environment): E)
  BEGIN
    !
    ! Terminates processing of an exception and reloads
    ! processor registers from the environment record E.
    !
    ! The PC, MOD, SP and FP fields of E must all be valid
    ! (as indicated by the validity bits) otherwise the
    ! program is terminated.
    !
    IF E.Validity & 2_10111 # 2_10111 THEN
      Abandon (CannotReloadEnvironment)
    END IF
    ProcessingException := FALSE
    ...
  END PROCEDURE

!
! What follows is the main procedure, called when a program exception
! occurs.  This may be a hard exception i.e. 32000 cpu trap, or a soft
! exception produced by the user program calling Signal or an X- type
! Panos procedure which failed.  This procedure executes using the
! small stack area reserved for exception processing.
!
PROCEDURE  ProcessException (INTEGER: Code,
                             RECORD(Environment): FailureEnvironment)
```

```
RECORD(Environment): ExceptionEnv, CurrentEnv
HandlerFn: Handler
INTEGER: Status

BEGIN
  !
  ! First check for recursive exception
  !
  IF ProcessingException THEN
    !
    ! Exception has occurred while another was being processed.
    ! This is fatal.
    !
    Abandon (ExceptionDuringExceptionProcessing)
  ELSE
    ProcessingException := TRUE
  END IF
  !
  ! Take a copy of the environment (i.e. all user-accessible
  ! CPU registers) at the point of failure.
  !
  ExceptionEnv := FailureEnvironment
  !
  ! Call handlers for active procedures in reverse of
  ! procedure-invocation order; if any of them handles
  ! the exception then control is returned to the program
  ! in a state defined by the (possibly updated)
  ! ExceptionEnv record.
  !
  CurrentEnv := ExceptionEnv
  Handler := ModuleHandler (CurrentEnv)
  Status := Handler (Exception, Code, CurrentEnv, ExceptionEnv)
  LOOP
    ResumeEnvironment (ExceptionEnv) IF Status = 1
    Status := Handler (Unwind, 0, CurrentEnv, NIL)
    EXIT IF Status # 1
    Handler := ModuleHandler (CurrentEnv)
    Status := Handler (ExceptionPassedOn, Code,
                       CurrentEnv, ExceptionEnv)
  END LOOP
```

```
    !
    ! If control reaches here then no handler was able to process
    ! the exception, so all that remains is to give some diagnostic
    ! information and stop the program.
    !
    IF HasBeenDescribed (Code) THEN
      !
      ! Say nothing if the Info field in the code indicates that
      ! a message has already been output describing the problem.
      !
    ELSE
      DiagnoseException (Code, FailureEnv)
      MarkAsHavingBeenDescribed (Code)
      DescribeEnvironment (FailureEnv)
    END IF
    Program.Stop (Code)
  END PROCEDURE
```

Module initialisation and termination

The handler may support facilities for module initialisation and/or termination. When these functions have been requested the handler is entered with CallType = Initialise or Stop.

Debugging Support

Call types Diagnose and DiagnosePassedon enable a handler to give a (possibly application specific) description of the actual exception. Call types DescribeFrame and DescribeModuleData are used to permit a language specific display (backtrace) of the state of a user's program at the time of an exception. This is intended to help the process of program development and debugging.

Formal Definition of Condition Handler

The formal specification of the handler is:

```
ConditionHandler(CARDINAL:CallType
        INTEGER:AdditionalParameter
        RECORD(ENVIRONMENT) REF:CurrentEnvironment
        RECORD(ENVIRONMENT) REF:ExceptionEnvironment);
        INTEGER:Result
```

The significance of Result is dependant on the CallType (detailed descriptions follow).

The reason for entry to the handler is indicated by the CallType code:

CallType	code	CallType	code
Initialise	0	DiagnosePassedOn	5
Stop	1	DescribeFrame	6
Exception	2	DescribeModuleData	7
ExceptionPassedOn	3	Unwind	8
Diagnose	4	Reserved	>8

The RECORD called ENVIRONMENT which holds state information has the format as shown in figure 2.

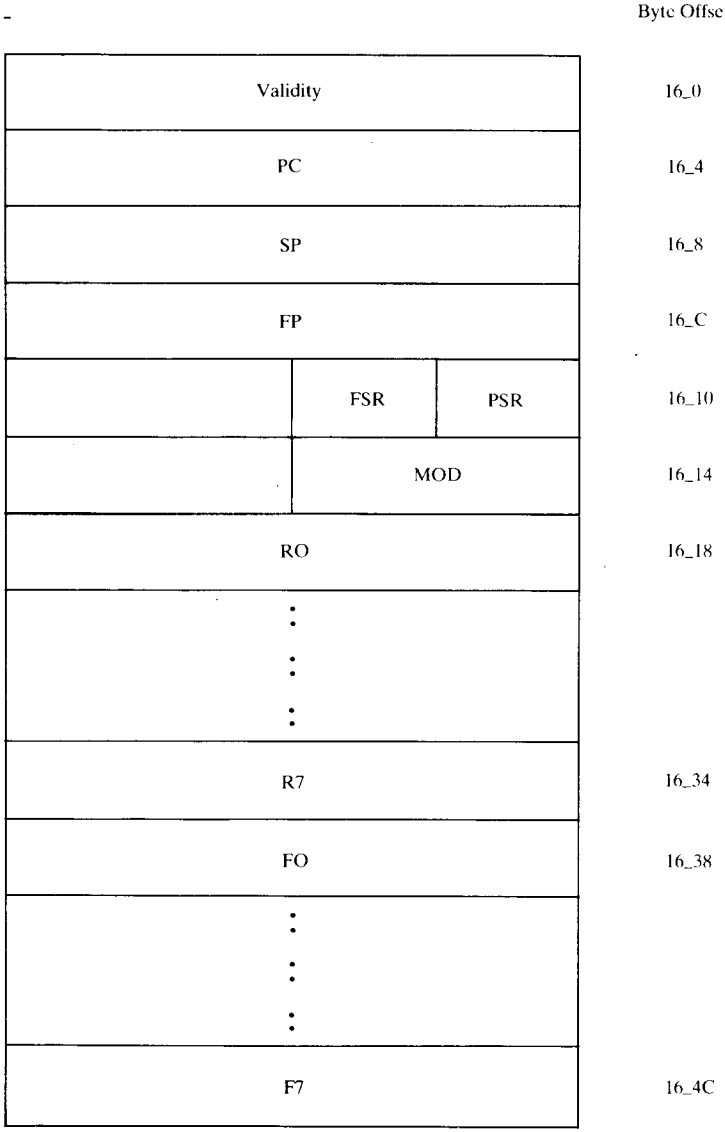


Figure 2 Environment record

The validity entry has bits set when later fields in the record are valid. The bits are assigned as in the table below (a field is valid when the corresponding bit is 1):

bit	field
0	PC
1	SP
2	FP
3	UPSR
4	MOD
5 - 12	R0 - R7
13 - 20	F0 - F7
21	FSR
22 - 31	Reserved

The various CallTypes are described in detail on the following pages. For each of the possible reasons for entry to the handler (i.e. for each CallType) an example of the actual call parameters is given together with the possible return values.

11.1 Initialise

Result:= ConditionHandler(Initialise,0, NIL, NIL)

Action

This call occurs once when the module is loaded. The handler should carry out any initialisation the module needs.

The order of call for initialisation corresponds to the order in which the modules are set up. For relocatable images this corresponds to the order of modules in the image. The Acorn 32000 Linker will order the modules in the same sequence as they are supplied, grouped into firstly modules from non-library files followed by modules from library files. Within each group modules are loaded in order of reading by the linker.

Call

No parameters other than CallType.

Return

Result $>=0$, initialisation completed.
<0, Error - initialisation not possible and Panos will stop the program with that error as result.

11.2 Stop

Result: = ConditionHandler(Stop,Status,NIL,NIL)

Action

This action is called for when the handler's module is being terminated. The handler should carry out any tidying up necessary.

This call will be made to any modules which have been called with type Initialise (in the same order as they were called for initialisation). If a module fails initialisation then only the modules up to and including the failing one are called.

Call

Status The status which will be returned from the program.

Return

The value returned has no significance.

11.3 Exception

Result:= ConditionHandler(Exception
ErrorCode
CurrentEnvironment
ExceptionEnvironment)

Result:= ConditionHandler(ExceptionPassedOn
ErrorCode
CurrentEnvironment
ExceptionEnvironment)

Action

An Exception has been detected.

If CallType = Exception then this is the first handler to be entered since occurrence of the exception, i.e. the exception occurred within this module. The handler may attempt to recover from the error, if this is possible. If CallType = ExceptionPassedOn then the exception has been passed on from another handler which could not handle the exception.

Call

ErrorCode The cause of the original exception.
< 0, Standard Panos Error Code. (including a buffer if the INFO field = 6).
> = 0, Application-Specific value (see description of Signal).

CurrentEnvironment

Pointer to an environment record describing the procedural frame currently being handled, i.e. one of the procedural frames in the active hierarchy stacked at the time the exception occurred. When CallType = Exception then the contents of the records referred to by CurrentEnvironment and ExceptionEnvironment are identical.

ExceptionEnvironment

Pointer to a record describing the machine state corresponding to the exception. For hardware exceptions the PC points at the failing instruction. For exceptions signalled by the Signal procedure PC points at the next instruction after the call to Signal. For X-type procedures which fail the PC points at the next instruction after the call to that procedure.

Return

Result

= 1, error handled. Instructs Panos to resume processing from state saved in ExceptionEnvironment record.
The state will be reloaded from the record according to the validity bits field. Any field whose validity bit is 0 will have the corresponding actual register loaded with 0. The registers PC, MOD, SP, FP must all be valid or the state cannot be reloaded (and the program will be abandoned).
= 0, error could not be handled.

11.4 Diagnose

```
Result:= ConditionHandler(Diagnose
             ErrorCode
             CurrentEnvironment
             ExceptionEnvironment)

Result:= ConditionHandler(DiagnosedPassedOn
             ErrorCode
             CurrentEnvironment
             ExceptionEnvironment)
```

Action

The handler is given an opportunity to produce a decription of the exception on the error stream. In particular this should be done for language and application specific errors.

Call

All parameters as for type Exception.

Return

Result = 1, diagnostics reported OK.
= 0, no output produced - Informs Panos to pass the call on, if possible, or produce diagnostics itself.

11.5 DescribeFrame

```
Result:= ConditionHandler(DescribeFrame
            Level
            CurrentEnvironment
            NIL)
```

Action

The handler is given an opportunity to produce textual information on the error stream describing the current procedural frame. Level is 0 when the CurrentEnvironment record describes the point of the exception, otherwise Level is a count of the number of DescribeFrame calls made so far.

Call

CurrentEnvironment

Pointer to an environment record.

Return

Result = 1, description given.
= 0, description not given. Panos will give a minimal description of the frame (PC, MOD values).

11.6 DescribeModuleData

```
Result:= ConditionHandler(DescribeModuleData
             0
             CurrentEnvironment
             NIL)
```

Action

The handler is given an oppurtunity to produce textual information on the error stream describing the static data associated with the CurrentEnvironment, i.e. in the module corresponding to its MOD field.

Call

CurrentEnvironment

Pointer to an environment record.

Return

Result = 1, description performed.
= 0, description not performed.

11.7 Unwind

```
Result:= ConditionHandler(Unwind
             0
             CurrentEnvironment
             NIL)
```

Action

The handler should update the CurrentEnvironment record to reflect the machine state that would be achieved after the procedure in the CurrentEnvironment procedural frame returns. As much of the environment as possible should be updated. The validity bits in the record should be set to indicate which fields are valid for the unwound state.

Call

CurrentEnvironment

Pointer to an environment record.

Return

Result = 1, unwind has occurred, the CurrentEnvironment now (at least partially) reflects the new state.
= 0, unwind has not been possible.

11.8 Reserved

```
Result:= ConditionHandler(Reserved
         0
         NIL
         NIL)
```

Action

Handlers should return 0 whenever the calltype is a reserved value.

Call

Return

Result =0, Action could not be performed.

11.9 Signal

Signal(INTEGER:Cause
ADDRESS:Buffer)

XSignal(INTEGER:Cause
ADDRESS:Buffer)

Action

Causes an exception to be signalled. If Cause is negative then the Cause value is interpreted as an error code (see Appendix A). If the value of the info field in the error code is 6, the Buffer parameter is copied to the saved image of R1, as seen in the environment record passed to the handler.

If Cause is positive then the buffer field is ignored.

Call

Cause	Single parameter to describe the exception.
Buffer	The address of a buffer to be passed to the exception handler containing extra information describing the exception.

Return

Depends upon the event handling in force (see previous sections).

11.10 CallHandler

```
CallHandler(CARDINAL:CallType
            INTEGER:AdditionalParameter
            RECORD(ENVIRONMENT) REF:CurrentEnvironment
            RECORD(ENVIRONMENT) REF:ExceptionEnviroment);
            INTEGER:Result

XCallHAndler(CARDINAL:CallType
            INTEGER:AdditionalParameter
            RECORD(ENVIRONMENT) REF:CurrentEnvironment
            RECORD(ENVIRONMENT) REF:ExceptionEnviroment);
            INTEGER:Result
```

Action

Gives the command specified by CallType to the handler associated with CurrentEnvironment.

Call

See formal definition of Condition Handler.

Return

Result As returned from the Handler.

Note XCallHandler is a synonym for CallHandler.

11.11 DeclareConditionHandler

DeclareConditionHandler(PROCEDURE: ConditionHandler);
INTEGER: Result;

XDeclareConditionHandler(PROCEDURE: ConditionHandler);

Action

Dynamically assign a module's condition handler.

Call

ConditionHandler
A 32000 external procedure descriptor referencing a procedure suitable to accept Condition Handler calls. This procedure will be installed as the condition handler of the calling module.

Return

Result $>=0$, Operation successful.
<0, Operation failed (= error code).

12. Asynchronous events

Asynchronous events are caused by interrupts occurring in the I/O processor. Examples of causes are keys being pressed, ESCAPE pressed and the user timer crossing zero. The procedures provided to control event processing reside in moduleHandler . The user program may declare one or more event handlers to deal with each type of asynchronous event. For each type of event the handler is installed using the procedure

DeclareEventHandler

and the handler is removed by a call to

RemoveEventHandler

Events are only signalled to an installed handler if they have been enabled by a call to the procedure

SetEventStatus,

which is also used to disable events.

The procedure EventStatus can be used to see whether an event is enabled or disabled.

When an event is signalled the handler is entered in user mode with the (user) stack pointer adjusted to point to a small private stack area. Interrupts are enabled.

Five parameters are supplied, the first of which gives the type of the event, the second and third being additional data (event specific), the fourth a handle to be used for identification purposes, and finally an environment record defining the machine state at the time of the event.

The formal specification of the handler is:

```
EventHandler(CARDINAL:EventCode
             CARDINAL:EventData1
             CARDINAL:EventData2
             CARDINAL:Handle
             RECORD(ENVIRONMENT) REF:Env)
```

A handler may be defined for each of the following events:

Event	Description	Code
0	Buffer empty	0
1	Buffer full	1
2	Keyboard interrupt	2
3	ADC conversion complete	3
4	Start of TV field pulse	4
5	Interval timer crossing 0	5
6	Escape condition detected	6
7	RS423 error	7
8	Network error	8
9	User event	9
128	Events lost	128
255	All other possible reasons.	

The host's (I/O processor's) X and Y values are passed to the handler in EventData1 and EventData2.

The procedures available for the control of events are described in the following pages. All these procedures reside in the module Handler.

For event 4 (TV vertical sync pulse) Event Data1 is the number of pulses since the event was last signalled.

Events are only signalled when the system is outside event handlers. Panos queues events waiting to be signalled until such time as they can be delivered. If the internal queues overflow then an event 128 is generated.

12.1 DeclareEventHandler

```
DeclareEventHandler(PROCEDURE:NewHandler
                    CARDINAL:Event
                    CARDINAL:Action
                    CARDINAL:Handle);
                    INTEGER:Result

XDeclareEventHandler(PROCEDURE:NewHandler
                    CARDINAL:Event
                    CARDINAL:Action
                    CARDINAL:Handle)
```

Action

Aranges for the given procedurre to be called upon the occurrence of a given asynchronous event.

NewHandler	New event handler.
Event	Determines which event.
Action	= 0, arrange for this handler to be called *after* all other existing handlers for this event have been called.
	= 1, arrange for this handler to be called *before* all other existing handlers for this event have been called.
	= 2, arrange for this handler *only* to be called and remove all other handlers from list.
Handle	This is passed to the handler when the event occurs.

Return

Result	=0, Operation successful. <0, Operation failed (=error code).

12.2 RemoveEventHandler

```
RemoveEventHandler(PROCEDURE:Handler
                   CARDINAL:Event
                   CARDINAL:Handle);
                   INTEGER Result

XRemoveEventHandler(PROCEDURE:Handler
                    CARDINAL:Event
                    CARDINAL:Handle)
```

Action

Removes the handler.

Call

Event	Determines which event.
Handler	Together with Handle this defines which instance of the handler is to be removed.
Handle	The handle asociated with a particular instance of the handler.

Return

Result	$>=0$, Operation successful. <0, Operation failed (=error code).

12.3 EventStatus

EventStatus(CARDINAL:Event);
INTEGER: Result

XEventStatus(CARDINAL:Event);
INTEGER: Result

Action

Return status about the event.

Call

Event Determines which event.

Return

Result
$>=0$, Operation successful.
$=0$, then event disabled.
$=1$, then event enabled.
<0, Operation failed ($=$error code).

12.4 SetEventStatus

SetEventStatus(CARDINAL:Event
BOOLEAN:Enable);
INTEGER:Result

XSetEventStatus(CARDINAL:Event
BOOLEAN:Enable);
INTEGER:Result

Action

Enable or disable the specified event and return status about the event.

Call

Event	Determines which event.
ENable	new status for the event.

Return

Result	> =0, Operation successful. =0, then event was previously disabled. =1, then event was previously enabled. <0, Operation failed (= error code).

13. Global String Variables

Global string variables are provided to facilitate communication across program boundaries.

Certain global strings are known by Panos and these provide a mechanism for defining the environment in which programs will run. For example, the string SYS$Version contains the Panos version identification.

All variables whose names begin with SYS$ are reserved and cannot be set. Some of the various global strings used by Panos are listed below. (The case of the variable name is not significant.)

SYS$Time — a string containing the system time (in textual form).

SYS$Date — a string containing the system date (in textual form).

SYS$Version — the system version string.

File$-<ext> — a string defining the transformation for file extension <ext>.

CLI$Result — result of last program to run. A string representation using base 16, e.g. -16_4 .

CLI$Path — defines command search order (see the *Panos Guide to Operations*).

CLI$Prompt — the string output by the CLI before reading commands. This string will be substituted by DecodeArg before being output.

CLI$Echo — controls the echoing of lines in command files (see section 3, *Panos Guide to Operations*)

Program$Verbosity
Program$Identify
Program$Abandon
Program$Confirm
Program$Force

these allow the inbuilt defaults for the corresponding utility switches to be overridden. For example if the user likes to

have programs identify themselves when they start then setting Program$Identify True will cause them all to output identification unless overridden by a -NoIdentify on the command line.

Program$Verbosity takes a cardinal value, the rest take a boolean. See description of ProgramVerbosityRequired for a description of the interpretation of the verbosity value.

Panos maintains a table in which are kept the global string names together with their associated string values. The case used in the string name is not significant. The interpretation of a global string value is not a function of the GlobalString module, but of the Panos modules and/or user programs which reference it.

Until it is deleted a Global String is visible by all software running under Panos.

A program may define its own strings and read strings set by other programs. This provides a simple method of communicating between user programs.

The procedures for control of the global strings are defined in the following pages. They reside in the moduleGlobalString .

13.1 SetGlobalString

SetGlobalString(STRING:GlobalStringName
STRING:GlobalStringValue);
INTEGER:Result

XSetGlobalString(STRING:GlobalStringName
STRING:GlobalStringValue)

Action

Adds the supplied name to the global string table and associates with it the supplied value.

Call

GlobalStringName

The name of the global string.

GlobalStringValue

A string value to be associated with the named global string.

Return

Result $>=0$, Operation successful.
<0, Operation failed (= error code).

13.2 GetGlobalString

GetGlobalString(STRING:GlobalStringName);
INTEGER:Result
STRING:GlobalStringValue

XGetGlobalString(STRING:GlobalStringName);
STRING:GlobalStringValue

Action

Looks up GlobalStringName in the global string table and returns the associated value. The search for GlobalStringName is case insensitive.

Call

GlobalStringName
The name of the GlobalString.

Return

Result $>=0$, Operation OK and GlobalStringValue is the value currently associated with the name.
<0, Operation failed (= error code).

13.3 DeleteGlobalString

DeleteGlobalString(STRING: StringName); INTEGER: Result

XDeleteGlobalString(STRING: StringName);

Action

Delete a global string variable from the Panos environment.

Call

StringName	The string variable to be deleted.

Return

Result	$>=0$, Operation successful. <0, Operation failed (= error code).

13.4 GetGlobalStringName

GetGlobalStringName (CARDINAL: Index);
INTEGER: Result
STRING: Name

XGetGlobalStringName (CARDINAL: Index);
STRING: Name

Action

Return the name of a Global string variable.

Call

Index — The index of the variable required. This can range between 0 and an upper bound determined by the number of variables defined. Thus to find all the names cycle index from 0 up in steps of 1 until an error is returned.

Return

Result — $>=0$, Operation successful.
<0, Operation failed (= error code).

14. Program Control

The procedures in moduleProgram control program invocation and execution.

When a program is invoked, the environment it has is:

IO: CurentInputStream
CurrentOutputStream
CurrentErrorStream
CurrentControlStream

all refer to the same streams as they did in the parent environment. 'Close' on these streams will have no effect on their state in the parent enviroment.

Store New enviroment, but obviously any memory in use before the program was invoked will be unavailable for allocation.

Event As in the environment of the invoker.

Global String Variables

As in the environment of the parent (since these are a single shared resource).

When the parent resumes after the program it invoked finishes, then the IO, Store and Event Environments are restored to their states before the invocation. Global Strings may have been changed.

14.1 Call

Call(STRING:FileName
STRING:ProcedureName
STRING:Arguments);
INTEGER:Result

XCall(STRING:FileName
STRING:ProcedureName
STRING:Arguments);
CARDINAL:Result

Action

Invoke a procedure from a file, passing it an argument string.

Call

FileName
The file containing the procedure.

ProcedureName
The procedure to be called.

Arguments
The argument string to be passed to the procedure.

Return

Result As returned from procedure or error code from 'Call'.

14.2 Run

```
Run(STRING:Name
        STRING:Arguments);
        INTEGER:Result

XRun(STRING:Name
        STRING:Arguments);
        CARDINAL:Result
```

Action

Invokes the relocatable program in the named file, passing it the given argument string.

Call

Name	The file containing the program in Relocatable Image Format (RIF). If no extension is given on the name then "-rif" is automatically supplied.
Arguments	The argument string passed to the program.

Return

Result	As returned from the program or error code from 'Run'.

14.3 Obey

Obey(STRING:CommandFileName
STRING:Arguments);
INTEGER:Result

XObey(STRING:CommandFileName
STRING:Arguments);
CARDINAL:Result

Action

Cause the given command file to be executed by the Command Interpreter of Panos.

Call

CommandFileName

The file to be executed. The extension "-cmd" is appended to the name if no other extension is present.

Arguments The arguments for the command file.

Return

Result As returned from the command file or error code from "Obey".

14.4 Invoke

Invoke(STRING:Name
STRING:Arguments);
INTEGER:Result

XInvoke(STRING:Name
STRING:Arguments);
CARDINAL:Result

Action

To invoke the named program, procedure, or command file from the set of known commands.

Call

Name	The name of a procedure, program or command file.
Arguments	Passed to the program, procedure or command file.

Return

Result	$>=0$, Result of program run. <0, Error in invocation or result of program run.

Note Invoke calls 'CallRunOrObey' with object name set to name.

14.5 CallRunOrObey

```
CallRunOrObey(STRING: Name
              STRING: ObjectName
              STRING: ArgumentString);
                    INTEGER: Result

XCallRunOrObey(STRING: Name;
                    STRING: ObjectName
                    STRING: ArgumentString);
                    CARDINAL: Result
```

Action

Determine the type of an object and either Call it, Run it or Obey it as appropriate.

Call

Name	The name of the object before it has been transformed (e.g. before any aliasing has been applied).
ObjectName	The name of the object to be called.
ArgumentString	The argument string to be supplied.

Return

Result	$>=0$, Operation successful.
	<0 , Operation failed (= error code).

Note the result may be that returned by the called object.

14.6 Name

Name(); INTEGER: Result
STRING: ProgramName

XName(); STRING: ProgramName

Action

Return the name by which a program was invoked.
(c.f. Name in CallRunOrObey.)

Call

No parameters.

Return

Result $>=0$, Operation successful.
<0, Operation failed (= error code).

14.7 FileName

FileName(): INTEGER: Result
STRING: ProgramName

XFileName(): STRING: ProgramName

Action

Return the name of the file from which the program was invoked.

Call

No parameters.

Return

Result $>=0$, Operation successful.
<0, Operation failed (= error code).

14.8 Stop

Stop(INTEGER:Result)

Action

Stops the current program and returns the given Result to the program which invoked it.

Call

Result	Result of the current program to be returned to the program which invoked it.

Return

This call *never* returns to the caller.

14.9 SetKnownCommandsPath

SetKnownCommandsPath(STRING:Path);
INTEGER:Result

XSetKnownCommandsPath(STRING:Path)

Action

Informs the program control module of the path to be used to determine the set of known commands. (This procedure is used by the command interpreter built in command ".newcommand".)

Call

Path — A comma separated list of Panos filenames, each name in this list can be either a directory or an aof library file (-lib extension).

Return

Result — $>=0$, Success, the new set has been defined.
<0, Error (=error code), no change has been made.

14.10 Arguments

Arguments();STRING:Arguments

XArguments();STRING:Arguments

Action

Informs the program of the arguments passed to it when it was invoked.

Call

No parameters.

Return

Arguments	The argument string passed to the program when it was invoked.

14.11 Verbosity

Verbosity();INTEGER:Result

XVerbosity();INTEGER:Result

Action

Inform utilities of the verbosity level as indicated by the current value of Program$Verbosity.

This call remains for compatibility with previous versions of Panos, new programs should use the VerbosityRequired procedure.

Call

No parameters.

Return

Result See VerbosityRequired.

14.12 IdentifyRequired

IdentifyRequired(HIDDEN:Handle
INTEGER:ErrorCode);
BOOLEAN:Result

XIdentifyRequired(HIDDEN:Handle
INTEGER:ErrorCode);
BOOLEAN:Result

Action

Return TRUE if the program should identify itself with Name and Version Number before performing its function.

Note that the global string 'Program$Identify' can be used to specify a system wide default for this call. If it has a value 'True' then IdentifyRequired() will return True unless -NoIdentify appeared in the argument string corresponding to the given handle.

Call

Handle	A handle returned from DecodeInit.
ErrorCode	Result returned from DecodeInit.

Return

Result	TRUE : Identification is required and should be written to un-reselected error stream. FALSE otherwise.

14.13 HelpRequired

HelpRequired(HIDDEN:Handle
INTEGER:ErrorCode);
BOOLEAN:Result

XHelpRequired(HIDDEN:Handle
INTEGER:ErrorCode);
BOOLEAN:Result

Action

Return TRUE if the program should give Help information. The normal program action should NOT occur if help is required.

Call

Handle	A handle returned from DecodeInit.
ErrorCode	Result returned from DecodeInit.

Return

Result	TRUE : Help is required, and should be written to the un-reselected ErrorStream. The program should then exit with Result = 0 without performing its normal function. FALSE otherwise.

14.14 SwitchRequired

SwitchRequired(STRING:Name
HIDDEN:Handle
BOOLEAN:Default);
BOOLEAN:Result

XSwitchRequired(STRING:Name
HIDDEN:Handle
BOOLEAN:Default);
BOOLEAN:Result

Action

Return the value required for the named switch argument. This is controlled by

a) specifying the name as a keyword in the argument string supplied to DecodeArg to produce the handle,
or
b) the value of the global string Program$<Name>, e.g. "Program$Force"
or
c) the value of the supplied default.

The table below gives the resulting value for all cases.

Argument String refers to the relevant portion only.

Argument String	Program$<Name>	Result
Not present	Not defined	Supplied Default
Not present	'True' or 'False'	Program$<Name> value
-No<Name>	don't care	False
-<Name>	don't care	True

Call

Name	The name of the keyword.
Handle	A handle returned from DecodeInit This is checked for validity and if invalid is ignored.
Default	The value of the default if no name is specified and no global string Program$<Name> exists.

Return

Result	The required value of the named switch.

14.15 VerbosityRequired

VerbosityRequired(HIDDEN:Handle);BOOLEAN:Result

XVerbosityRequired(HIDDEN:Handle);BOOLEAN:Result

Action

Return the value required of the verbosity level for the current program. This is controlled by

a) the use of -Verbose, -NoVerbose, or -Verbosity n in the argument string supplied to DecodeArg to produce the handle,
or b) the value of the global string Program$Verbosity,
or c) the value of the default i.e. 3.

The table below gives the resulting value for all cases. Argument String refers to the relevant portion only.

Argument String	Program$Verbosity	Result
Not present	Not defined	3
Not present	n	n
-NoVerbose	don't care	0
-Verbose	don't care	3
-Verbosity n	don't care	n

Call

Handle — A handle returned from DecodeInit. This is checked for validity and if invalid is ignored, 0 is a suitable invalid value if required.

Return

Result — The required verbosity.

15. Command line interpreter

The procedures described in this chapter give access to the facilities of the Panos command interpreter. They reside in moduleCommand . For details of the user-interface behaviour of the command line interpreter, see the *Panos Guide to Operations.*

15.1 InterpretString

InterpretString(STRING:CommandLine);
INTEGER:Result

InterpretString(STRING:CommandLine);
CARDINAL:Result

Action

The CommandLine is passed to the Panos command interpreter and if it is valid it is interpreted.

Call

CommandLine	The command for the interpreter.

Return

Result	> = 0, then operation successful. < 0, then operation failed (= error code).

15.2 InterpretCommands

InterpretCommands(); INTEGER: Result
XInterpretCommands(); CARDINAL: Result

Action

Call the command interpreter to interpret a stream of commands from the current control stream.

Call

No parameters.

Return

Result As returned by the invoked CLI.

16. Wild symbol expansion

These procedures are in moduleWild . Instead of typing in the whole of a file or directory name, the user can substitute names, or parts of names, with symbols representing groups of characters. These are shown below. See also section 7.17.

? Matches any single character.
* Matches any zero or more characters, within a name

16.1 Match

Match (STRING: wild
STRING: name);
INTEGER: Result
BOOLEAN: matched

XMatch (STRING: wild
STRING: name);
BOOLEAN: matched

Action

Perform a simple case-insensitive wildcard match of a string against a template.

Call

wild (Case-insensitive) wildcarded template.

Permitted wildcards are:
'?' - matches any 1 character
'*' - matches 0 or more characters.

name String to test for match against template.

Return

Result >=0, Operation successful: matched=TRUE if wild matches name.
<0, Operation failed (=error code).

16.2 Replace

Replace (STRING: wild
STRING: match
STRING: template);
INTEGER: Result
STRING: Substituted

XReplace (STRING: wild
STRING: match
STRING: template);
STRING: Substituted

Action

Substitutes wildcard instantiations from a wild match into a second wild template, to create a new string. Wildcard correspondences are made left to right, separately for '?' and '*'.

Call

wild	(Case insensitive) wildcarded template.
match	String matching wild template.
template	Template for result string, having same number of wildcards as wild template.

Return

Result	$>=0$, Operation successful. substituted holds new string.
	<0, Operation failed (=error code).

17. BBC Library

These procedures are in moduleBBC . They provide access to the host's operating system and should only be used as a last resort, i.e. when the required operation is not provided by another procedure in the Panos library. The integrity of Panos is not guaranteed following the use of these procedures.

17.1 OSByte

```
OSByte(CARDINAL:ByteNo
       CARDINAL:Param1
       CARDINAL:Param2);
       CARDINAL:Result1
       CARDINAL:Result2
       BOOLEAN:CBit
       INTEGER:Result

XOSByte(CARDINAL:ByteNo
        CARDINAL:Param1
        CARDINAL:Param2);
        CARDINAL:Result1
        CARDINAL:Result2
        BOOLEAN:CBit
```

Action

Invokes the miscellaneous set of OSBYTE system calls on the Host.

Call

ByteNo	The osbyte call type.
Param1	Least significant byte is X value.
Param2	Least significant byte is Y value.

Return

Result	$>=0$, Operation successful. Result1 is BBC X value. Result2 is BBC Y value. CBit is BBC C flag. <0, Operation failed (= error code).

17.2 OSWord

```
OSWord(CARDINAL:OSWdno
       ADDRESS:ParamBlock);
       INTEGER:Result

XOSWord(CARDINAL:OSWdno
        ADDRESS:ParamBlock)
```

Action

Invokes the miscellaneous set of OSWORD system calls on the Host.

Call

OSWdNo	number of osword request
ParamBlock	Pointer to osword control block

Return

Result	> = 0 , Operation successful and ParamBlock holds OSWord return values. < 0 , Operation failed (= error code).

17.3 OSFile

```
OSFile(CARDINAL:OSFileNo
       STRING:Name,
       ADDRESS:ParamBlock);
       INTEGER:Result

XOSFile(CARDINAL:OSFileNo
        STRING:Name
        ADDRESS:ParamBlock)
```

Action

The BBC operating system primitive OSFILE is called.

Call

OSFileNo	Number of osfile request.
Name	File name.
ParamBlock	Record containing load address, execution address, data start address and data end address.

Return

Result	$>= 0$, Operation successful and ParamBlock holds OSFILE return values. < 0 , Operation failed (= error code).

Appendix A Panos-generated Errors

The definitive set of Panos errors will be found in the !Perror file supplied as part of the Panos release.

Hardware errors

02	address translation trap
03	floating point error trap
04	illegal instruction trap
05	supervisor call trap
06	divide by zero trap
07	flag trap
08	breakpoint trap
09	trace trap
0a	undefined instruction

Data conversion errors

00	Bad base
01	Bad number string
02	Number not in specified base
03	Overflow

Store management errors

00	Environment overflow
01	Too many pop environments
02	Tag overflow
03	Bad heap tag
04	Stack and heap overlap
05	Not enough free store
06	Request for zero length block
07	Not an allocated block
08	Bad address for split block
09	Failed to allocate module space
0a	Operation not available for module table block
0b	No space for new environment

IO errors

00	No saved environment
01	No stream selected
02	Bad stream number
03	Bad device name '%'
04	Bad device data '%'
05	Illegal operation
06	Not enough streams
07	Bad open type
08	Filing system not found
09	End of file
0a	Printer not available
0b	No default filing system
0c	No workspace
0d	Invalid filing system
0e	Not implemented
0f	Implementation failure
10	No buffer space
11	Printer in use
12	Tab string too big
13	Device removed
14	Duplicate devicename '%'

Loader errors

00	Invalid image
01	Module '%' not found
02	Code symbol '%' not found
03	Data symbol '%' not found

Time and date errors

00	Time not set
01	Invalid binary time
02	Bad time string

Exception handling errors

00	No handler installed

Event handling errors

00	Illegal event
01	Illegal action

Global string errors

00	Buffer too small
01	Environment variable '%' not found
02	Can't set '%' - reserved variable
03	No room
04	Can't delete '%' - reserved variable

Program control errors

00	No current commands
01	Unable to set new command path
02	% not found
03	No CLI path
04	Escape
05	Invalid result code
06	Invalid stop code

Argument decoding errors (keystring)

00	Bad information
01	Type mismatch
02	Index greater than number of arguments
03	Bad index parameter
04	Keyword '%' not known
05	Missing keyword delimiter
06	Missing keyword name
07	Word too large

08	Missing slash option
09	Conflicting type specification
0a	Bad number in quantity
0b	Too large a quantity
0c	Cannot have quantity zero
0d	State can only have quantity one
0e	-Help and -Identify wanted but bad arguments
0f	Duplicated options
10	Missing close square bracket
11	Missing character after escape
12	Too many defaults
13	Missing trailing quote
14	No defaults with state keywords

Argument decoding errors (user arguments)

16	No defaults with exact quantity
17	Too many arguments for '%'
18	Cannot attach '%' to a keyword
19	Argument '%' expects a value
1a	Argument '%' needs a parameter
1b	Too few arguments for '%'
1c	Bad integer argument %
1d	Bad cardinal argument %
1e	'%' is not a keyword
1f	-Help wanted but bad arguments
20	-Identify wanted but bad arguments
21	Substitution of <% > not possible
22	Missing substitution bracket
23	String result buffer too small
24	Minus substitute only with existent
25	Abbreviation '%' is ambiguous
27	Rest can only have one argument
29	File '%' not found
2a	'%' does not match any files

Filing errors

00	% is not datestamped
01	Bad filing system
02	% is a directory
03	File '%' not found
04	Can't rename across filing systems
05	Bad information handle
06	Not enough workspace
07	Bad directory index
08	Bad filing system name : %
09	Directory '%' not found
0a	Bad date stamp
0b	File name '%' is illegal
0c	Null file name
0d	Attempt to rename across drives
0e	File '%' already exists
0f	Too many wild paths
10	Bad template
11	No wild match
12	File '%' too large for buffer
13	Bad file name '%'

Error module errors

00	Not an error code
01	No error information available

Wild pattern matching errors

00	Too many star fields : %
01	Inconsistent template : %
02	No match : %
03	Buffer too small

Appendix B

Bibliography

Panos Technical Reference Manual

BBC Microcomputer System User Guide

Acorn Computers Limited
Scientific Division
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
Telephone 0223 245200