

OLIB REFERENCE

Version 2.30

(C) Copyright Psion PLC 1990-98

All rights reserved. This manual and the programs referred to herein are copyrighted works of Psion PLC, London, England. Reproduction in whole or in part, including utilization in machines capable of reproduction or retrieval, without express written permission of Psion PLC, is prohibited. Reverse engineering is also prohibited.

The information in this document is subject to change without notice.

Psion and the Psion logo are registered trademarks, and Psion, Psion MC, Psion HC, Psion Series 3, Psion Series 3a and Psion *Workabout* are trademarks of Psion PLC.

TopSpeed is a registered trademark of Clarion Software Corporation. Intel 8086 and 80286 are registered trademarks of Intel Corporation. IBM, IBM XT and IBM AT are registered trademarks of International Business Machines Corp. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Apple and Macintosh are registered trademarks of Apple Computer Inc. VAX and VMS are registered trademarks of Digital Equipment Corporation. Brief is a registered trademark of Underware Inc. Psion PLC acknowledges that some other names referred to are registered trademarks.

CONTENTS

1 Introduction.....	1-1
Using OLIB classes	1-2
Notation	1-2
Names	1-2
Method function prototypes	1-2
The ★ symbol.....	1-3
Long parameters.....	1-3
Class diagrams	1-3
Class hierarchy.....	1-3
Structured Error Recovery.....	1-4
Use of the p_leave mechanism.....	1-4
Panic numbers.....	1-5

2 The ROOT Class.....	2-1
Class definition	2-1
Property	2-1
ROOT methods	2-2
Destroy the instance	2-2

3 The TIME Class.....	3-1
Precursors	3-2
Class definition	3-2
Property	3-3
TIME methods	3-3
Set time.....	3-3
Sense time.....	3-4
Add seconds	3-4
Add days	3-5
Add months.....	3-5
Add years	3-5
Sense format	3-6
Set format	3-6
Get system date and time information.....	3-7

4 The SGBUF Segmented Buffer Class.....	4-1
Precursors	4-1
Class definition	4-2
Property	4-2
SGBUF methods	4-2
Destroy.....	4-2
Initialise.....	4-2
Sense data by position	4-3
Insert.....	4-3

Delete.....	4-3
Extract	4-4
Compress	4-4
Count characters	4-4
Sense previous characters	4-4
Allocate segments	4-4

5 Variable Array Classes..... 5-1

Precursors	5-1
Class diagram	5-1
Usage summary.....	5-2
Record pointers	5-2
VAROOT.....	5-3
Class definition	5-3
Property	5-4
VAROOT methods.....	5-4
Destroy.....	5-4
Count records.....	5-4
Append a record.....	5-4
Insert a record	5-4
Delete a record	5-5
Set key for comparisons.....	5-5
Compare two records by pointer	5-5
Compare two records by number	5-5
Sort	5-6
Find (binary chop).....	5-6
Insert in sequence.....	5-6
Search	5-6
Reset	5-7
Replace a record.....	5-7
Deferred VAROOT methods	5-7
Copy a record	5-7
Get record length.....	5-7
Swap two records	5-7
Initialise	5-8
Set capacity	5-8
Compress	5-8
Delete sequence of records.....	5-8
Insert sequence of records.....	5-8
Point to record.....	5-9
Point to record data	5-9
VAFIX.....	5-9
Class definition	5-10
Property	5-10
VAFIX methods.....	5-10
Replace a record.....	5-10
Copy a record	5-10
Get record length.....	5-10
Swap two records	5-10
VASTR.....	5-11
Class definition	5-12
Property	5-12
VASTR methods.....	5-12
Initialise.....	5-12
Set capacity.....	5-13
Compress	5-13
Delete a sequence of records.....	5-13
Insert a sequence of records.....	5-13
Get record length.....	5-13
Point to record.....	5-13
Point to record data	5-14
Copy a record	5-14

VAFLAT	5-14
Class definition	5-14
Property	5-15
VAFLAT methods	5-15
Initialise	5-15
Set capacity	5-15
Compress	5-15
Delete sequence of records.....	5-15
Insert sequence of records.....	5-15
Point to record.....	5-16
Point to record data	5-16
VASEG.....	5-16
Class definition	5-17
Property	5-17
VASEG methods.....	5-17
Initialise.....	5-17
Set capacity	5-17
Compress	5-17
Delete a sequence of records.....	5-17
Insert a sequence of records.....	5-18
Point to record.....	5-18
Point to record data	5-18
VAXVAR	5-18
Class definition	5-19
Property	5-19
VAXVAR methods	5-20
Initialise.....	5-20
Compare two records by pointer	5-20
Delete a sequence of records.....	5-20
Insert a sequence of records.....	5-20
Replace record.....	5-21
Copy a record.....	5-21
Get record length.....	5-21
Point to record data	5-21
VAXVARS	5-21
Class definition	5-22
Property	5-22
VAXVARS methods - see VAXVAR methods	5-22

6 Editable Documents 6-1

Document content	6-1
Addressable character positions.....	6-1
Usage	6-1
Precursors	6-2
Class diagram	6-2
EPROOT.....	6-2
Class definition	6-3
Property	6-3
EPROOT methods.....	6-3
Set content	6-3
Scan by word.....	6-4
Count words.....	6-4
Scan by paragraph.....	6-4
Count paragraphs.....	6-4
Scan by block	6-4
Append a paragraph	6-5
Copy whitespace indentation	6-5
Copy range to front of clipboard	6-5
Copy range to back of clipboard	6-5
Insert from clipboard.....	6-6
Modify characters in a range	6-6
Copy text to buffer.....	6-6
Set document capacity	6-6

Deferred EPROOT methods	6-6
Initialise	6-6
Sense document length	6-6
Sense characters forwards.....	6-6
Sense characters backwards.....	6-6
Insert characters.....	6-6
Copy out characters	6-6
Delete characters	6-6
Clear the document	6-6
Compress allocated storage.....	6-6
EPFLAT	6-7
Class definition	6-7
Property	6-7
EPFLAT methods	6-8
Destroy.....	6-8
Initialise.....	6-8
Sense document length.....	6-8
Sense characters forwards.....	6-8
Sense characters backwards.....	6-8
Insert characters.....	6-8
Copy out characters	6-9
Delete characters	6-9
Clear the document	6-9
Compress allocated storage.....	6-9
Set document capacity.....	6-9
Set buffer granularity.....	6-9
Sense start of data	6-9
EPSEG.....	6-10
Class definition	6-10
Property	6-10
EPSEG methods.....	6-10
Initialise.....	6-10
Sense document length.....	6-11
Sense characters forwards.....	6-11
Sense characters backwards.....	6-11
Insert characters.....	6-11
Copy out characters	6-11
Delete characters	6-11
Clear the document	6-11
Compress allocated storage.....	6-11

7 Resource Files 7-1

Precursors	7-1
Class definition	7-1
Property	7-2
RSCFILE methods	7-2
Destroy.....	7-2
Initialise.....	7-2
Allocate buffer and read resource	7-2
Read resource.....	7-2

8 Binary File Management 8-1

Precursors	8-1
Class diagram	8-1
BFILE	8-2
Class definition	8-2
Property	8-2
BFILE methods.....	8-3
Destroy.....	8-3
Open file.....	8-3
Close file.....	8-3

Read from file.....	8-3
Set buffer size	8-3
Sense record data.....	8-4
Reposition to start.....	8-4
TLVFILE	8-4
Class definition	8-5
Property	8-5
TLVFILE methods.....	8-6
Open TLV file.....	8-6
Reposition to start	8-6
Count records	8-6
Write a record	8-6
Set current record.....	8-6
Sense current record number	8-7
Delete a record	8-7
Replace a record.....	8-7
Read record of specific type(s).....	8-7
TLVDATA	8-8
Class definition	8-8
Property	8-8
TLVDATA methods	8-9
Open and read file.....	8-9
Save data to file.....	8-9
Check if changed.....	8-9
Process a record read from a file.....	8-10
Get a record to be saved.....	8-10
Reset all data.....	8-10
Deferred TLVDATA methods	8-10
Set the TLV file characteristics.....	8-10
Set in-memory data for a record	8-10
Sense in-memory data for a record	8-10
SERFILE	8-11
Class definition	8-11
Property	8-12
SERFILE methods	8-13
Reset all data.....	8-13
Set the serial file characteristics.....	8-13
Set in-memory data for a record	8-13
Sense in-memory data for a record	8-14

9 The CLEANUP Class..... 9-1

Precursors	9-1
Class diagram	9-2
Class definition	9-2
Property	9-2
CLEANUP Methods.....	9-3
Destroy.....	9-3
Initialise list	9-3
Add item	9-3
Remove item	9-3
Delete item.....	9-3
Delete all items at current level	9-4
Set cleanup level	9-4
CLEANUP convenience functions.....	9-4
Add an item	9-4
Add an object	9-4
Add an I/O channel	9-5
Add an allocated cell	9-5
Add a shared allocated cell.....	9-5
Add a DYL	9-5
Remove an item.....	9-6
Delete an item	9-6

10 The APPMAN Application Manager Class..... 10-1

Active object priorities.....	10-2
Active object scheduling.....	10-3
The ao_run return value.....	10-3
Precursors	10-4
Class diagram	10-4
Class definition	10-4
Property	10-5
APPMAN Methods	10-5
Initialise.....	10-5
Wait on I/O semaphore.....	10-6
Start scheduler	10-6
Stop scheduler	10-9
Add a task.....	10-9
Load a resource	10-9
Load a resource to a buffer	10-10
Generate resource file name	10-10
Display notifier	10-10
Notify an error.....	10-11
Clean up resources and report an error	10-11
Find application image file.....	10-11
Ensure only one copy running	10-12
Change active object priority	10-12

11 The ACTIVE Class and Active Objects 11-1

Precursors	11-1
Class definition	11-2
Property	11-2
ACTIVE methods	11-3
Destroy the instance	11-3
Initialise the instance.....	11-3
Make a request to run	11-3
Cancel a request to run	11-3
Handle an error	11-4
Process an event.....	11-4

12 Idle Objects and the AIDLE Class 12-1

Idle objects.....	12-1
AIDLE	12-1
Precursors	12-2
Class diagram	12-2
Class definition	12-2
Property	12-2
AIDLE methods.....	12-2
Initialise.....	12-2
Run	12-2
Examples	12-3

13 Timer Active Object Classes..... 13-1

Class diagram	13-1
Precursors	13-1
TIMER	13-2
Class definition	13-2
Property	13-2
TIMER methods.....	13-2
Initialise.....	13-2
Queue (relative).....	13-2

Queue (absolute).....	13-3
ANIMATOR.....	13-3
Class definition	13-3
Property	13-4
ANIMATOR methods	13-4
Initialise.....	13-4
Run	13-4
BUZSND	13-4
Class definition	13-5
Property	13-5
BUZSND methods	13-5
Initialise.....	13-5
Cancel.....	13-6
Queue.....	13-6
Run	13-6

14 File Active Objects..... 14-1

Precursors	14-1
Class diagram	14-1
FACTIVE	14-1
Class definition	14-2
Property	14-2
FACTIVE methods	14-2
Initialise.....	14-2
Cancel.....	14-2
Handle error	14-3
Close any open file	14-3
FSCAN	14-3
Class definition	14-4
Property	14-4
FSCAN methods	14-5
Directory read	14-5
Process read completion	14-5
Close directory files.....	14-6
Match a found name.....	14-6
Start a directory scan	14-6
Deferred FSCAN methods.....	14-7
Scan completion	14-7
Next directory name	14-7
Next file name.....	14-7
End of subdirectory	14-8
FNODE.....	14-8
Class definition	14-9
Property	14-9
FNODE methods.....	14-9
Read from appropriate channel.....	14-9
Process read completion	14-9
Close both open channels	14-10
Start list generation	14-10
Deferred FNODE methods	14-10
Handle completed list.....	14-10
Process a list item.....	14-10
FCASY	14-11
Class definition	14-11
Property	14-12
FCASY methods	14-12
Cancel request.....	14-12
Read request.....	14-12
Write request.....	14-12
Process read or write completion	14-13
Close the file	14-13
Open a file	14-13

Deferred FCASY methods.....	14-14
Inform of completion.....	14-14
FCSYNC.....	14-14
Class definition	14-14
Property	14-14
FCSYNC methods.....	14-15
Read request.....	14-15
Write request.....	14-15

15 File Lists 15-1

Precursors	15-1
Class diagram	15-1
PSELVAR.....	15-2
Class definition	15-2
Property	15-2
PSELVAR methods.....	15-3
Compare two records by pointer	15-3
PNODE.....	15-3
Class definition	15-3
Property	15-3
PNODE methods.....	15-4
Handle error	15-4
Handle completed list.....	15-4
Process a list item.....	15-4
PSEL	15-4
Class definition	15-5
Property	15-6
PSEL methods.....	15-7
Initialise.....	15-7
Handle error	15-7
Cancel list building	15-7
Add a file name.....	15-8
Add a directory name	15-8
Process end of a scan.....	15-8
Build file name list.....	15-8
Ascend one subdirectory level.....	15-9
Descend to a subdirectory	15-9
Set a new path.....	15-9
Sense a file list item	15-9
Select a node list entry.....	15-9
Ascend to the drives level.....	15-10
Set/clear a file tag.....	15-10
Get a tagged file.....	15-10
Set the file list order	15-10
Deferred PSEL methods	15-11
Process the completion of list building.....	15-11

16 File Management Classes..... 16-1

Precursors	16-1
Class diagram	16-1
Class definition	16-3
Property	16-3
FMAN methods	16-4
Initialise the file manager	16-4
Cancel an outstanding request	16-4
Copy files	16-5
Delete files	16-5
Rename files.....	16-6
Make a directory tree.....	16-7
Delete a directory structure.....	16-7

Copy a device	16-7
Format a device	16-8
Name a device	16-8
Set file attributes	16-8
Deferred FMAN methods	16-9
Processing completed	16-9
Processing a new file	16-10
Section of processing completed	16-11
File exists	16-11
Determine error response	16-11
FMMK	16-12
Class definition	16-12
Property	16-12
FMMK methods	16-12
Process a make directory request	16-12
FMFMT	16-13
Class definition	16-13
Property	16-13
FMFMT methods	16-13
Process a format request	16-13
Process formatting error	16-14
FMSCAN	16-14
Class definition	16-14
Property	16-14
FMSCAN methods	16-15
Process an error	16-15
Next file name	16-15
Next directory name	16-16
End of subdirectory	16-16
Scan completion	16-16
FMSRC	16-17
Class definition	16-17
Property	16-17
FMSRC methods	16-17
Process a read completion	16-17
FMTARG	16-18
Class definition	16-18
Property	16-18
FMTARG methods	16-18
Process a write completion	16-18
File manager example	16-19

17 The LOCS Local File Scan Class..... 17-1

Precursors	17-1
Class definition	17-2
Property	17-2
LOCS methods	17-3
Perform a file scan	17-3
Check file name match	17-3
Deferred LOCS methods	17-3
Process a file name	17-3

18 System Services	18-1
Precursors	18-1
SYSTEM	18-1
Class definition	18-2
Property	18-2
SYSTEM methods	18-2
Initialise	18-2
Control icon positioning	18-2
Set link paste server	18-3
Get link paste server	18-3
Run an application by file name	18-3
19 Inter-process Communication.....	19-1
Precursors	19-1
Inheritance tree	19-1
IPCS	19-2
Class definition	19-2
Property	19-2
IPCS methods	19-3
Destroy.....	19-3
Initialise	19-3
Queue a message read	19-3
Cancel read request	19-3
Process a message	19-3
Handle error	19-4
Add item to server queue.....	19-4
SERVER.....	19-4
Class definition	19-5
Property	19-5
SERVER methods	19-5
Destroy.....	19-5
Initialise	19-5
Handle error	19-5
Deferred SERVER methods	19-6
Process a message	19-6
20 Link Paste.....	20-1
Precursors	20-2
Class diagram	20-2
LINKCL	20-2
Class definition	20-2
Property	20-2
LINKCL methods.....	20-3
Destroy.....	20-3
Initiate a transaction.....	20-3
Request data	20-3
Terminate a transaction.....	20-3
LINKSV	20-4
Class definition	20-4
Property	20-4
LINKSV methods.....	20-5
Initialise	20-5
Process a message	20-5
Deferred LINKSV methods	20-6
Set data format	20-6
Provide data	20-6

CHAPTER 1

INTRODUCTION

This manual is a reference document for Psion's OLIB library. It provides a comprehensive guide to the library and documents the classes, methods, properties, inheritance hierarchies and other information essential for understanding and using the library. It assumes familiarity with the concepts of Object Oriented Programming.

The *Object Oriented Programming Guide* is a useful pre-requisite as it provides the necessary background to Object Oriented Programming as implemented at Psion. It can, of course, be read in conjunction with the *OLIB Reference* manual.

The OLIB library is supplied as the *olib.dyl* dynamic link library in the ROM of all SIBO machines. It contains a collection of classes, built on the services of the PLIB library.

OLIB classes provide a range of services that are independent of the user interface used by an application. For example, they include a number of classes for creating array type objects which possess sophisticated behaviour.

Use of the OLIB library allows complex applications to be built quickly and reliably. The OLIB object classes can be used directly or can be subclassed by any application code. They are both subclassed and used directly as components by the user interface libraries (for example, HWIM).

Each chapter in this manual contains a description of either a single class or a number of closely related classes. For example, the *Variable Array Classes* chapter describes a number of classes that implement different types of array, each with a variable number of elements, while *The TIME Class* chapter describes a single class.

The description of each class follows the same format. It includes the purpose of the class, the hierarchical relationship of the class to other classes, the actual class definition, a description of the property and a complete list and discussion of the methods. References to relevant manuals are included if any pre-requisite information is needed.

The first chapter contains a description of the `ROOT` class, from which all other classes are derived. It is, therefore, a required class in all object oriented programs.¹

OLIB object classes provide services which include:

- data storage in a segmented buffer
- arrays with a variable number of elements
- basic text editing
- event management and scheduling
- time management and timers
- file management
- inter-process communication

¹ It is, however, permissible for a category that has no intrinsic dependence on other OLIB classes to define its own root class and thereby eliminate all dependency on OLIB. See, for example, the *Building a Dynamic Library* chapter of the *Object Oriented Programming Guide*.

Using OLIB classes

An application (or DYL) that either subclasses or creates an instance of an OLIB class must declare an external reference to the OLIB library in its category file. If, for example, an application's category file has the name *myprog.cat*, the content of this category file must start with the following lines:

```
IMAGE myprog

EXTERNAL olib
```

This ensures that, amongst other things, the defined constants representing the external category numbers for the OLIB category (in this case `CAT_MYAPP_OLIB`) is available to application code. (It is, in any case, required in virtually all category files, to provide access to the `ROOT` class, from which all other classes are derived - see the later *ROOT class* section in this chapter.)

In the source code of the MYPROG application, an instance of an OLIB class - say, of `VASEG` - would be created with `p_new` (or `f_new`) as follows:

```
p_new(CAT_MYPROG_OLIB, C_VASEG);
```

If *myprog.cat* defines a subclass of an OLIB class (say, the class `SUBVASEG`) this would exist in the local category. An instance is created using the local category number `CAT_MYPROG_MYPROG`, as follows:

```
p_new(CAT_MYPROG_MYPROG, C_SUBVASEG);
```

Similar considerations apply to instances created by means of `f_newsend`.

Notation

Throughout this manual, all references to the Series 3 should be taken to refer to the Series 3a and the Workabout, unless explicitly stated otherwise.

Names

Except in class diagrams, a class name is always given in upper case, for example `VAROOT`.

The method name in the title line of the description of each method is the defined symbol for the method number, without its leading `o_`. In the body of the text this name, in lower case letters, is used to refer to the *method function* (or more simply, the *method*) whereas the upper case name refers to the corresponding *message*. Thus, an object's `destroy` method function is executed when the object receives a `DESTROY` message.

Method function prototypes

The description of each method contains a function prototype that specifies the nature of any return value and the parameters with which the method is called. The parameters exclude the object handle and the method number.

For example, a method for the class `VAFLAT` with the title line:

VA_TEST	Compare two records by pointer
----------------	---------------------------------------

and prototyped as:

```
VOID va_test(UBYTE *prec1, UBYTE *prec2);
```

would be invoked by:

```
p_send4(hand, o_VA_TEST, prec1, prec2);
```

where `hand` is the handle of an object of the class in question.

This corresponds to a method function declared in C source code as:

```
METHOD VOID vaflat_va_test(PR_VAFLAT *self, UBYTE *prec1, UBYTE *prec2)
{
    ...
}
```

The ☼ symbol

The enter and leave mechanism (which uses `p_enter` and `p_leave`) is commonly used to implement structured error recovery. See the *Error Handling and Error Recovery* chapter of the *Object Oriented Programming Guide* and the *Error Handling* chapter of the *PLIB Reference* manual.

Some methods (the vast majority of destroy methods, for example) can never fail and will therefore never call `p_leave`. The title line of a number of the more significant methods of this type are marked with a leading ☼ symbol.

With the enter and leave mechanism, a call to `p_leave` should only occur within the protection of a `p_enter` harness. If `p_leave` is called outside a `p_enter` harness, the process will be panicked with panic number 47.

The `p_leave` mechanism and its use in method functions is discussed briefly in the section on *Structured Error Recovery* later in this chapter.

Long parameters

A small number of OLIB class methods require a `LONG` or a `ULONG` parameter. For the reasons explained in the *Introduction* chapter of the *Object Oriented Programming Guide*, the message-sending mechanism in TopSpeed C does not support such parameters and they should be passed as two `INT` (or `UINT`) parameters, where the first is the least significant word and the second is the most significant word of the data. In such a case the actual method prototype is always followed by a conceptual form, illustrating the intent of the parameters.

Class diagrams

To illustrate the inheritance and using relationships between classes, most chapters will contain at least one class diagram.

The notation is a subset of that used by Grady Booch and described in his book *Object-oriented Analysis and Design with applications* (2nd edition) with two minor changes;

- classes which are referenced, but not described, within a chapter (i.e. classes whose full description lies in other chapters of this manual or in a different manual), are underlined,
- the diagrams do not distinguish between 'has' (aggregation) and 'using' (client/supplier) relationships.

Also note that ultimate inheritance from the root class is assumed and is not shown.

Class hierarchy

In understanding the structure of a specific class, remember that methods and property are often inherited from a superclass (or superclasses).

While a class may contain new methods and property, it may also re-define methods inherited from a superclass (or superclasses). Note that methods in a superclass can be what are known as *deferred* methods.

To help illustrate these relationships, each class description in this manual is accompanied by a diagram which shows that class and its superclass(es) in hierarchical order. This diagram is placed at the beginning of the class description.

The diagram consists of a series of adjacent columns. The rightmost column represents the class being described and will be marked by a double line border while the column to its left represents its immediate superclass (if any) marked by a single line border and so on up the hierarchy. Each column is headed by the class name followed by two boxes; the first lists that class's property and the second lists its methods.

Deferred methods are separated from the preceding methods by a blank line and are printed in italics. If a class re-defines an inherited method, the method name in the appropriate superclass is written with a line through it.

For example, the following diagram would be included in a description of class CCCC subclassed from BBBB which itself is subclasses AAAA.

AAAA	BBBB	CCCC
property_1 property_2	property_3	property_4 property_5
method_a ~~method_b~~	~~method_b~~ ~~method_c~~ method_d method_e	method_b method_c method_x method_y method_z <i>method_u</i>

In this illustration, `method_a` is supplied by the superclass AAAA, while `method_b` is replaced in class BBBB and further replaced in class CCCC. Another method, `method_c`, is introduced in class BBBB but replaced in CCCC, and so on. Note that `method_u` is a deferred method.

The ROOT class from which all classes are derived is assumed and will not be shown in the diagrams.

Structured Error Recovery

As mentioned earlier, the enter and leave mechanism is commonly used to implement structured error recovery.

Use of the `p_leave` mechanism

In general, you should assume that all methods NOT marked with the ✱ symbol (as discussed in the section on Notation) are capable of calling `p_leave`, even if this is not explicitly mentioned in the method description. In some cases, such as where a method calls, directly or indirectly, a deferred method (which is supplied by a subclasser) it is not possible to specify whether the method may result in `p_leave` being called.

In the event of an error (such as out of system memory) occurring a method may:

- call `p_leave`, passing the (negative) error number,
- return the error number,
- either call `p_leave` or return an error number, depending on the nature of the error.

Some methods call `p_leave(0)`, which has the effect of returning from the `p_enter` harness (with the return value zero) without signalling an error. This is used, for example, to provide a normal exit from a deeply nested function call, without the need for a zero return value to be passed back through the chain of calls. Intermediate functions in the chain may then be declared as `VOID`.

Some method functions that may call `p_leave` are declared as `VOID`. One reason for this may be that the method forms part of a chain, as described in the preceding paragraph. If user code were to send such a message within a `p_enter` harness, the value returned from `p_enter` would be indeterminate if no error arose. The solution is to construct a shell function which sends the message and then returns zero, and call this shell within a `p_enter` harness. The call to `p_enter` will then return either zero (if the method calls `p_leave(0)` or it executes to completion) or a negative error number.

Panic numbers

See the *Error Handling and Error Recovery* chapter of the *Object Oriented Programming Guide* and the *Error Handling* chapter of the *PLIB Reference* manual for a discussion of panics and panic numbers.

OLIB panics a client that attempts an illegal operation, using the following panic numbers:

130	Record in flat variable array is out of range
131	Number of record to insert new record before in flat VA is out of range
132	Attempt to set capacity of flat VA less than current number of records
133	Number of record to delete in flat VA is out of range
134	Record in segmented VA is out of range
135	Outside range of segmented buffer
136	Tried to delete outside segmented buffer
137	Record in string VA whose address is sought is out of range
138	Attempt to set capacity of string VA less than current number of records
139	Number of record to insert new record before in string VA is out of range
140	Number of record to delete in string VA is out of range
141	Did not read correct number of bytes from resource file
142	Image fails to contain built-in resource file
143	Stray signal death in Application Manager
144	Request to clear area outside character map
145	Bad type/length binary file record header
146	Read on serial port already outstanding or read buffer not allocated
147	Serial port read buffer too small for requested read
148	Write to serial port already outstanding or write buffer not allocated
149	Serial port write buffer too small for requested write
150	Read on serial port outstanding when tried to see no. of characters available to read
151	Read on serial port outstanding when tried to flush serial read buffer
152	Read or write outstanding when tried to set serial port characteristics
153	Control to set/get not supported
154	OPL translator invoked with empty command line
155	Unrecognised code for setting the console
156	String passed to consol is too long
157	IPCS Message read failed
158	Stray signal death in IPCS server list

CHAPTER 2

THE ROOT CLASS

ROOT
pc
destroy

The `ROOT` class is the ultimate superclass from which all other classes are derived. It provides the basic behaviour which defines a class as being an Object Oriented entity. It contains property which is, in effect, the "hook" by which the operating system can keep hold of an instance of a class.

A class *must* be subclassed from either an existing class or the `ROOT` class.

Class definition

Defined in category file *olib.cat* (generated header file *olib.g*).

```
CLASS    root
The ultimate superclass - all other classes have root as their ancestor.
{
  ADD destroy
  PROPERTY
  {
    P_OBJECT pc;    class link
  }
}
```

Property

`root.pc`

The class link. This should not be accessed by any subclass. This is a data structure of type `P_OBJECT` which is defined in `p_std.h`.

The `P_OBJECT` structure is as follows:

```
typedef struct
{
  HANDLE hcat;
  HANDLE hclass;
} P_OBJECT;
```

where `hcat` and `hclass` are `HANDLE` (a synonym for `INT`) data types and have the same meaning as the first two words of the `P_CLASS` data structure.

For more information on the underlying mechanisms of the Object Oriented system, see the *Object Oriented Programming* chapter of the *PLIB Reference* manual and/or the *Introduction* chapter and the appendices of the *Object Oriented Programming Guide*.

ROOT methods

J DESTROY

Destroy the instance

`VOID destroy(VOID);`

Use the EPOC O/S `LibDestroy` system service to destroy the instance, together with all component objects that are marked for automatic destruction (by means of a `PROPERTY n` declaration in the class definition).

The `destroy` method of every subclass must, ultimately, execute `ROOT`'s `destroy` method, either by calling the `root_destroy` method function directly or, more usually, by supersending a `DESTROY` message to the `ROOT`.

CHAPTER 3

THE TIME CLASS

TIME
ds f
to_set to_sense to_add_years to_add_months to_add_days to_add_secs to_set_format to_sense_format to_get_sysdat

The main purpose of the `TIME` class is to convert from one representation of time to another (where the word *time* is used in the general sense, to include the date and the time of day).

The `TIME` class stores a *current time* within its property. Conversion is done by setting the time in one representation with the `to_set` method and then retrieving the same time in a different representation with the `to_sense` method.

The `TIME` class "adds value" to the basic PLIB functions by:

- using its stored state (the current time) to provide a more convenient interface to changing between the three PLIB representations of time (system time, `P_DAYSEC` and `P_DATE`)
- converting to and from textual representations of time (this is not provided by the PLIB functions)

As well as the methods associated with conversion, there are methods to add a signed number of years, months, days and seconds to the current time.

The time object stores a current time within its property in a `P_DAYSEC` struct which contains:

- the number of days since January 1st 1900 (day 0 is January 1st)
- the number of seconds in the day

The number of days is stored in a long and can represent dates over a range of about 11.7 million years from year 1900. This range of validity is larger than any of the other representations supported.

The property also stores format information that modifies the textual representations of time. A subset of the format information is used to interpret textual representations of time.

Although the method descriptions which follow show only English time and date text, the actual text is language-dependent. (See the *Language and country* section of the *General System Services* chapter of the *PLIB Reference manual*.)

See also `HTIME`, the `HWIM` subclass of `TIME`.

Precursors

A knowledge of the PLIB/EPOC time and date functions will aid the understanding of the `TIME` class. The chapter *Time, Timers and Dates* in the *Plib Reference* manual contains a description of the basic PLIB/EPOC time and date functions and the various formats in use.

Class definition

The `TIME` class subclasses `ROOT` and is defined in the sub-category file *time.cl* (with generated header file *time.g*).

```
CLASS time root
{
  ADD to_set          Set the time in a variety of representations
  ADD to_sense        Sense the time in a variety of representations
  ADD to_add_years     Add years to the current time
  ADD to_add_months    Add months to the current time
  ADD to_add_days      Add days to the current time
  ADD to_add_secs      Add seconds to the current time
  ADD to_set_format    Set the format for string representations of time
  ADD to_sense_format  Sense the format for string representations of time
  ADD to_get_sysdat    Get day/month/suffix/ampm name or system format

  CONSTANTS
  {
    SET_TIME_SECONDS      0
    SET_TIME_DATE         1
    SET_TIME_DAYSEC       2
    SET_TIME_NOW          3
    SET_TIME_DATESTR      4
    SET_TIME_TIMESTR      5
    SENSE_TIME_SECONDS    0
    SENSE_TIME_DATE       1
    SENSE_TIME_DAYSEC     2
    SENSE_TIME_STRING     3
    SENSE_TIME_DATESTR    4
    SENSE_TIME_TIMESTR    5
    SENSE_TIME_FIELDS     0x8000
    ! Field types
    FLD_TIME_DAY          0
    FLD_TIME_MONTH        1
    FLD_TIME_YEAR         2
    FLD_TIME_HOUR         3
    FLD_TIME_MINUTE       4
    FLD_TIME_SECOND       5
    FLD_TIME_DAYNAME      6
    ! String format masks
    PR_TIME_DDMMYY        0x0000
    PR_TIME_MMDDYY        0x0001
    PR_TIME_YYMMDD        0x0002
    PR_TIME_DATE_ORDER    0x0003
    PR_TIME_NO_DAY        0x0004
    PR_TIME_NO_MONTH      0x0008
    PR_TIME_NO_YEAR       0x0010
    PR_TIME_MONTH_NAME    0x0020
    PR_TIME_SUFFIX_NAME   0x0040
    PR_TIME_DAY_NAME      0x0080
    PR_TIME_NO_CENTURY    0x0100
    PR_TIME_NO_SECS       0x0200
    PR_TIME_AMPM          0x0400
    TY_TIME_DAY           0
    TY_TIME_MONTH         1
    TY_TIME_SUFFIX        2
    TY_TIME_AMPM          3
    TY_TIME_FORMAT        4
    ! Buffer capacities for strings, including terminating zero
    LN_TIME_DAY_NAME      14  Guaranteed max buffer size for a day name
    LN_TIME_MONTH_NAME    14  Guaranteed max buffer size for a month name
    LN_TIME_DATE_STR      48  Guaranteed max buffer size for a date
    LN_TIME_TIME_STR      12  Guaranteed max buffer size for a time
  }
}
```

```

TYPES
{
    typedef struct
    {
        UWORD flags;           date and time format
        UBYTE dsep;           date separator
        UBYTE tsep;           time separator
    } SE_TIME_FORMAT;
    typedef struct
    {
        UWORD fld;           field type FLD_TIME_XXX
        TEXT *buf;           address of field
        UWORD len;           length of field
    } SE_TIME_FIELD;
    typedef union
    {
        TEXT *t;
        ULONG *l;
        P_DAYSEC *ds;
        P_DATE *dt;
    } PT_TIME_DATA;           Address of time data
    }
PROPERTY
{
    P_DAYSEC ds;           days and seconds
    SE_TIME_FORMAT f;       text format
}
}

```

Property

time.ds	the currently set date and time for an instance, in days and seconds, stored as two LONGs. It is manipulated by those methods adding or subtracting seconds, days, months and years. It should not be changed by any subclass.
time.f	the currently set textual format and the date and time separator characters that will be used when requesting the time and date as a zero terminated text string. It should not be directly accessed by any subclass.

TIME methods

J TO_SET

Set time

INT to_set(INT format, VOID *pdata);

Set TIME's current date and time from a variety of representations of the time. The representation is specified by format as follows:

SET_TIME_SECONDS	pdata is the address of a ULONG containing the system time. The system time is the number of seconds since 00:00:00, January 1st 1970.
SET_TIME_DATE	pdata is the address of a P_DATE structure. See the PLIB manual for a description of the P_DATE structure.
SET_TIME_DAYSEC	pdata is the address of a P_DAYSEC structure. See the PLIB manual for a description of the P_DAYSEC structure.
SET_TIME_NOW	uses p_date to set the time to the current system time (pdata is ignored).
SET_TIME_DATESTR	pdata is the address of a string representation of the date (the time of day is not changed). The date string should be in numeric form and include the day, month and year number in the current order (as defined in the format, set using the TO_SET_FORMAT method) and delimited by the current date delimiter or any punctuation character. If the year field has two digits and is greater than or equal to 70, it is added to 1900; otherwise it is added to 2000. If a date is set successfully, the method returns the number of characters processed.

SET_TIME_TIMESTR *pdata* is the address of a string representation of the time of day (the date is not changed). The time string should contain either two or three fields, separated by the current time delimiter or any punctuation character. If there are two fields, they are assumed to be hours and minutes and the seconds are set to zero. If there is a trailing string which matches the am/pm string returned by `TO_GET_SYSDAT`, this is processed with appropriate effect. If a time is set successfully the method returns the number of characters processed.

If successful (i.e. **pdata* defined a legal time), the method returns either zero or, if appropriate, the positive number of characters processed. Otherwise the current time is not modified and the method returns one of the negative error numbers `E_GEN_ARG` or `E_GEN_FAIL`.

J TO_SENSE

Sense time

`INT to_sense(INT format, VOID *pdata);`

Write a partial or complete representation of the current time to **pdata*, where the representation depends on *format* as follows:

SENSE_TIME_SECONDS	<i>pdata</i> is the address of a <code>ULONG</code> to take the current time in system time format.
SENSE_TIME_DATE	<i>pdata</i> is the address of a <code>P_DATE</code> structure to take the current time.
SENSE_TIME_DAYSEC	<i>pdata</i> is the address of a <code>P_DAYSEC</code> structure to take the current time.
SENSE_TIME_STRING	<i>pdata</i> is the address of a buffer, which must be at least <code>LN_TIME_DATE_STR</code> bytes in length, to receive a textual representation of the date and time as a zero terminated string. The conversion is controlled by the current format, as last set by the <code>to_set_format</code> method.
SENSE_TIME_DATESTR	<i>pdata</i> is the address of a buffer, which must be at least <code>LN_TIME_DATE_STR</code> bytes in length, to take a textual representation of the date as a zero terminated string.
SENSE_TIME_TIMESTR	<i>pdata</i> is the address of a buffer, which must be at least <code>LN_TIME_DATE_STR</code> bytes in length, to take a textual representation of the time of day as a zero terminated string.

See the `to_get_sysdat` method for further information on the size limits for various components of the date and time strings.

Returns zero if the time was written successfully to **pdata*, or one of the following negative error numbers:

E_GEN_UNDER	if the current time is too early for the requested format
E_GEN_OVER	if the current time is too late for the requested format

J TO_ADD_SECS

Add seconds

`INT to_add_secs(INT lsw, INT msw);`
`INT to_add_secs(LONG nsecs);` (*conceptual*)

Add the signed quantity *nsecs* to the current time of day. The `LONG nsecs` is actually passed in the message as two `INT` parameters, *lsw* (least significant word) and *msw* (most significant word).

If the number of seconds added is such as to cross the end (or start, if *nsecs* is negative) of a day the number of days is adjusted accordingly.

The new time of day is set modulo 86400 (the number of seconds in a day).

Returns zero if the adjusted time and date is legal. Otherwise the current time and date are not changed and the method returns one of the following negative error numbers;

E_GEN_UNDER	if <i>nsecs</i> is negative and it would take the time earlier than January 1st 1900
E_GEN_OVER	if <i>nsecs</i> is positive and it would take the time later than the latest date which can be supported (about 11.7 million years AD)

For example, to add one minute to the current time:

```
INT AddMinute(PR_TIME *self)
{
    return(p_send4(self, O_TO_ADD_SECS, 60, 0));
}
```

J TO_ADD_DAYS

Add days

```
INT to_add_days(INT lsw, INT msw);
INT to_add_days(LONG ndays); (conceptual)
```

Add the signed quantity `ndays` to the current date. The current time of day is unaffected. The `LONG ndays` is actually passed in the message as two `INT` parameters, `lsw` (least significant word) and `msw` (most significant word).

Returns zero if the new date is legal. Otherwise the current date is not changed and the method returns one of the following negative error numbers;

<code>E_GEN_UNDER</code>	if <code>ndays</code> is negative and it would take the time earlier than January 1st 1900
<code>E_GEN_OVER</code>	if <code>ndays</code> is positive and it would take the time later than the latest date which can be supported (about 11.7 million years AD)

For example:

```
INT AddDays(PR_TIME *self, LONG ndays)
{
    INT lsw, msw;

    lsw=ndays&0xffff;
    msw=ndays>>16;
    return(p_send4(self, O_TO_ADD_DAYS, lsw, msw));
}
```

J TO_ADD_MONTHS

Add months

```
INT to_add_months(INT nmonths);
```

Add the signed quantity `nmonths` to the current date. The current time of day is unaffected.

Returns zero if the new date is legal. Otherwise the current date is not changed and the method returns one of the following negative error numbers;

<code>E_GEN_UNDER</code>	if <code>nmonths</code> is negative and it would take the time earlier than January 1st 1900
<code>E_GEN_OVER</code>	if <code>nmonths</code> is positive and it would take the time later than the latest date which can be supported (about 11.7 million years AD).

J TO_ADD_YEARS

Add years

```
INT to_add_years(INT nyears);
```

Add the signed quantity `nyears` to the current date. The current time of day is unaffected.

Returns zero if the new date is legal. Otherwise the current date is not changed and the method returns one of the following negative error numbers;

<code>E_GEN_UNDER</code>	if <code>nyears</code> is negative and it would take the time earlier than January 1st 1900
<code>E_GEN_OVER</code>	if <code>nyears</code> is positive and it would take the time later than the latest date which can be supported (about 11.7 million years AD)

J TO_SENSE_FORMAT**Sense format**

```
VOID to_sense_format(SE_TIME_FORMAT *pf);
```

Write the current format data to *pf. See the to_set_format method for an explanation of the fields in the SE_TIME_FORMAT struct.

It is typically used to obtain the current settings before changing a format field by means of the to_set_format method.

J TO_SET_FORMAT**Set format**

```
VOID to_set_format(SE_TIME_FORMAT *pf, UINT mask);
```

Stores format parameters which are subsequently used when converting to and from textual representations of time.

If pf is NULL, the method uses system services to set as many components of the format as possible. It does this by sending itself a TO_GET_SYSDAT message with a format of TY_TIME_FORMAT. In this case the value of mask is ignored.

Otherwise, pf should point to an SE_TIME_FORMAT struct.

The values of pf->dsep and pf->tsep should be the new character codes for the required date separator and time separator. Either (or both) may be NULL, in which case the corresponding existing separator is not changed.

The value of pf->flags, together with mask, sets or clears a combination of format flags. Both should contain an ored combination of the following bit flags, where the bits set in mask determine which items should be changed, and the corresponding bit in pf->flags (set or clear) determines the new value. To change all bits as specified by pf->flags, mask should be set to 0xff. The bit flags have the following meanings in pf->flags:

PR_TIME_DDMMYY	if present, the date will be written in day-month-year order (European style)
PR_TIME_MMDDYY	if present, the date will be written in month-day-year order (USA style)
PR_TIME_YYMMDD	if present, the date will be written in year-month-day order (Japanese style - also good for sorting)
PR_TIME_NO_DAY	if present, the day is omitted
PR_TIME_NO_MONTH	if present, the month is omitted
PR_TIME_NO_YEAR	if present, the year is omitted
PR_TIME_MONTH_NAME	if present, the month is shown as a name rather than a number
PR_TIME_SUFFIX_NAME	if present, a suffix is added to the day number (eg 1st, 2nd, 3rd).
PR_TIME_DAY_NAME	if set, the day name is written (with a trailing comma) before the date
PR_TIME_NO_CENTURY	if present, the year is displayed in two digits without the century
PR_TIME_NO_SECS	if present, the time is displayed without a seconds field
PR_TIME_AMPM	if present, the time is written in the 12 hour system with a trailing "am" or "pm" (preferred in the USA), otherwise, it is written using the 24 hour system

If setting PR_TIME_DDMMYY, PR_TIME_MMDDYY or PR_TIME_YYMMDD, no more than one of them should be present in pf->flags and all three should be set in mask. (PR_TIME_DDMMYY is zero so, strictly speaking, it does not need to be set in mask. Setting the other two in mask and not including any of them in pf->flags has the same effect as including PR_TIME_DDMMYY. For this reason, the constant PR_TIME_DATE_ORDER is defined as a combination of PR_TIME_MMDDYY and PR_TIME_YYMMDD.)

The following example first sets default format data from the current system settings. It then modifies the date separator character to a colon (:) and adjusts the time format to include a display of seconds and an am/pm indicator:

```
VOID TimeSetup(PR_TIME *self)
{
    UINT mask;
    SE_TIME_FORMAT f;

    p_send4(self, O_TO_SET_FORMAT, NULL, 0); /* set defaults */
    f.dsep=':';
    f.tsep=NULL; /* don't change this */
    f.flags=PR_TIME_AMPM;
    mask=PR_TIME_NOSECS | PR_TIME_AMPM;
    p_send4(self, O_TO_SET_FORMAT, &f, mask); /* clear NOSECS bit and set AMPM bit */
}
```

J TO_GET_SYSDAT Get system date and time information

INT to_get_sysdat(UBYTE *buf, UINT type, UINT n);

Unless type is TY_TIME_FORMAT, write a time-related name, as a zero terminated string, to *buf and return the length of the string copied.

The caller is responsible for ensuring that the buffer is of sufficient size to take the appropriate string. Regardless of the language, each string is guaranteed not to exceed the following lengths:

- a day name will not exceed LN_TIME_DAY_NAME (14) characters
- a month name will not exceed LN_TIME_MONTH_NAME (14) characters
- any format of date string (built from a combination of items, including those written by this method) will not exceed LN_TIME_DATE_STR (48) characters
- any format of time string (built from a combination of items, including those written by this method) will not exceed LN_TIME_TIME_STR (12) characters

The name type is selected by the value of type, as follows:

TY_TIME_DAY	selects the name of the day corresponding to the day number, n (modulo 7)
TY_TIME_MONTH	selects the name of the month where n is the month number (modulo 12)
TY_TIME_SUFFIX	selects the day suffix name where n is the day in month number (modulo 31)
TY_TIME_AMPM	selects the am/pm string where n is 0 for am and 1 for pm (modulo 2)

Although this method may be of use to a client, it is primarily present for internal use when formatting date and time strings.

It uses PLIB and operating system services to get the names. A subclass can replace this method if alternative names are required.

If type is TY_TIME_FORMAT, the method writes a system-supplied SE_TIME_FORMAT structure to *buf (with no terminating zero) and returns sizeof(SE_TIME_FORMAT).

The TY_TIME_FORMAT type is used by TO_SET_FORMAT when the address of the format data is NULL. Although this type does not really fit with the others, its inclusion here localises the system-dependent portion of the time class to this method.

CHAPTER 4

THE SGBUF SEGMENTED BUFFER CLASS

SGBUF
hd nbytes cur
destroy sb_init sb_point sb_insert sb_delete sb_extract sb_compress sb_count sb_backpoint sb_allocseg

Conceptually, the data held in an instance of the SGBUF class can be regarded as being stored in a variable sized linear buffer.

The data is actually stored in memory in a linked list of allocated heap cells of equal size. Insertions and deletions may allocate or free cells and will, in general, cause data to be transferred from one cell to another. All cells will generally be at least 50% full.

Since the segmentation of the data is largely hidden from a user of SGBUF, the data should not be accessed other than via the supplied methods.

Precursors

An understanding of the SGBUF class will be aided by a knowledge of:

- the PLIB memory allocator functions.
- the p_enter and p_leave error handling services.

Class definition

The SGBUF class subclasses ROOT and is defined in the sub-category file *varray.cl* (with generated header file *varray.g*).

```
CLASS sgbuf root
Segmented buffer object
{
  REPLACE destroy
  ADD sb_init          Initialise with segment length
  ADD sb_point         Get the address from a position
  ADD sb_insert        Insert at specified position
  ADD sb_delete        Delete at specified position
  ADD sb_extract       Extract from specified position
  ADD sb_compress      Compress buffer
  ADD sb_count         Return no. of bytes in buffer
  ADD sb_backpoint     Get the address before a position
  ADD sb_allocseg      Allocate memory segments as required
  TYPES
  {
    typedef struct
    {
      P_QUE q;          Segment header
                        Links to neighbouring segments
      UWORD len;        Number of bytes currently in segment
    } PR_SGBUF_HD;
    typedef struct
    {
      PR_SGBUF_HD *seg; Current segment, or NULL
      UWORD base;      Character position of start of current segment
      UWORD ofs;       Current offset into segment
    } PR_SGBUF_SBO;
  }
  PROPERTY
  {
    PR_SGBUF_HD hd;      Head of queue
    UWORD nbytes;       Total number of bytes in buffer
    PR_SGBUF_SBO cur;   Current position for efficient positioning
  }
}
```

Property

<code>sgbuf.hd</code>	The head of the linked list of segments containing the data. Also contains the length of each allocated segment.
<code>sgbuf.nbytes</code>	The total number of bytes of content.
<code>sgbuf.cur</code>	The last accessed position as the current segment buffer, the character position of the first character in that buffer and the character offset within the buffer. This is used internally for efficient positioning when scanning sequentially.

SGBUF methods

J DESTROY

Destroy

```
VOID destroy(VOID);
```

Delete the content, freeing all of the allocated segments, and then supersend the DESTROY message.

J SB_INIT

Initialise

```
VOID sb_init(UINT len);
```

Initialise the segment queue and set the required length of a segment by setting `sgbuf.hd.len` to `len`. Note that this is the length to be made available for data. The amount of memory allocated for a segment will actually be `sgbuf.hd.len` plus the length of the header (i.e. the length of `PR_SGBUF_HD`).

The choice of the value of `len` is a compromise that depends on the nature of the data that is to be stored.

A small value reduces the potentially wasted space within each segment (at worst a segment may be only half full) but increases the likelihood that data will need to be moved from one segment to another during insertion or deletion. A small segment size is therefore more appropriate when the data is not expected to change very frequently.

A larger value of `len` is more suitable for situations where the data will frequently change, but may result in more potentially wasted space, particularly if the maximum expected content is small. In addition, when data *does* move from segment to segment, the larger the segment, the more data is likely to be moved.

As a rough guideline, it may be noted that all text editing applications on the Series 3 use a segment size of 64 bytes. If necessary in a particular case, an optimum value can be found empirically by timing a typical operation with a range of different segment sizes.

When a sequence of fixed length items is to be stored, insertion, deletion and access to items will be much more efficient if the segment size is an exact multiple of the item size.

No segments are allocated until data is inserted.

J SB_POINT

Sense data by position

```
UINT sb_point(VOID **pbuf, UINT pos);
```

Write, to `*pbuf`, the address of the data at position `pos`.

Calls `p_panic(P_PANIC_P_SGBUF_1)` if `pos` is outside the range of the segmented buffer content.

Updates its internal record of the current position.

Returns the number of contiguous bytes of data available at `*pbuf`.

Does not write to `*pbuf` and returns zero if there is no data in the segmented buffer.

SB_INSERT

Insert

```
VOID sb_insert(UINT pos, VOID *pbuf, UINT len)
```

Create a gap of size `len` bytes at position `pos` in the segmented buffer and then inserts the `len` bytes of data from `pbuf`.

The inserted data may span more than one segment, with more segments being allocated as necessary. If opening the gap causes data to overflow from the segment containing the insertion point, then this data will be inserted into the last of any newly created segments and/or any immediately following segment that previously existed.

The allocation of segments is performed by sending an `SB_ALLOCSEG` message.

The value of `sdbuf.nbytes` is adjusted to indicate the new number of bytes in the segmented buffer.

Calls `p_panic(P_PANIC_P_SGBUF_1)` if position `pos` is outside the range of the segmented buffer.

Inserts nothing and calls `p_leave(E_GEN_NOMEMORY)` if it fails to allocate any extra segments required.

J SB_DELETE

Delete

```
VOID sb_delete(UINT pos, UINT len);
```

Delete `len` bytes from position `pos`. This may involve copying data between segments. If a segment becomes empty then it will be freed.

The value of `sdbuf.nbytes` is adjusted to indicate the new number of bytes in the segmented buffer.

Calls `p_panic(P_PANIC_P_SGBUF_1)` if position `pos` is outside the range of the segmented buffer. Calls `p_panic(P_PANIC_P_SGBUF_2)` if position `pos+len` is outside the range of the segmented buffer.

J SB_EXTRACT**Extract**

```
VOID sb_extract(UINT pos, VOID *buf, UINT len);
```

Copy `len` bytes of data from position `pos` into the buffer at `buf`. If necessary, data is copied from more than one segment. The buffer is assumed to be large enough to hold `len` bytes of data.

Calls `p_panic(P_PANIC_P_SGBUF_1)` if position `pos` is outside the range of the segmented buffer.

J SB_COMPRESS**Compress**

```
VOID sb_compress(VOID);
```

Compress the segmented buffer by moving data to fill the early segments. Any segments that are emptied by this process will be freed.

J SB_COUNT**Count characters**

```
UINT sb_count(VOID);
```

Return the number of bytes currently held within the segmented buffer.

J SB_BACKPOINT**Sense previous characters**

```
UINT sb_backpoint(VOID **pbuf, UINT pos);
```

Write, to `*pbuf`, the address of the byte of data that is the lowest in memory and contiguous with the byte at position `pos-1`. In general, `*pbuf` will contain the address of the first byte of data in the segment containing the byte at position `pos`. If position `pos` is at the beginning of a data segment, `*pbuf` contains the address of the first byte of data in the previous data segment.

Returns the number of bytes between the position corresponding to `*pbuf` and position `pos`. If the return value is `n`, data is only guaranteed to be valid at addresses of `*pbuf` to `*pbuf+n-1` inclusive.

If `pos` is zero nothing is written to `*pbuf` and the method returns zero. This is the only position for which the return value is zero.

Calls `p_panic(P_PANIC_P_SGBUF_1)` if position `pos` is outside the range of the segmented buffer. This method may be regarded as the complement of `sb_point` and would typically be used when searching backwards through the content.

SB_ALLOCSEG**Allocate segments**

```
INT sb_allocseg(PR_SGBUF_HD *pseg, UINT nseg);
```

Allocate a chain of `nseg` empty segments and insert the chain after the segment at `*pseg`. Each allocated segment is initialised to be empty by setting the `len` field of the segment header to zero. The amount of memory required for each segment is `sgbuf.hd.len` *plus* the length of the segment header (i.e. the length of `PR_SGBUF_HD`).

This method is used internally and is not intended to be called by a user of the `SGBUF` class.

Does not allocate any segments and calls `p_leave(E_GEN_NOMEMORY)` if there is not enough memory to allocate all `nseg` segments.

CHAPTER 5

VARIABLE ARRAY CLASSES

The classes described in this chapter implement arrays of a variable number of records, in which records are referenced by number. The first record is record zero and the last record is record $n-1$, where n is the total number of records in the array. Depending on the particular class, records may be of fixed or variable length.

Unlike static C arrays, the space for the array is dynamically allocated from the heap. This means that adding a record to an array can fail owing to a failure to allocate additional memory. For some subclasses, it is possible to pre-set the capacity; this facility may be used when the required capacity is known in advance in order to avoid out of memory failures.

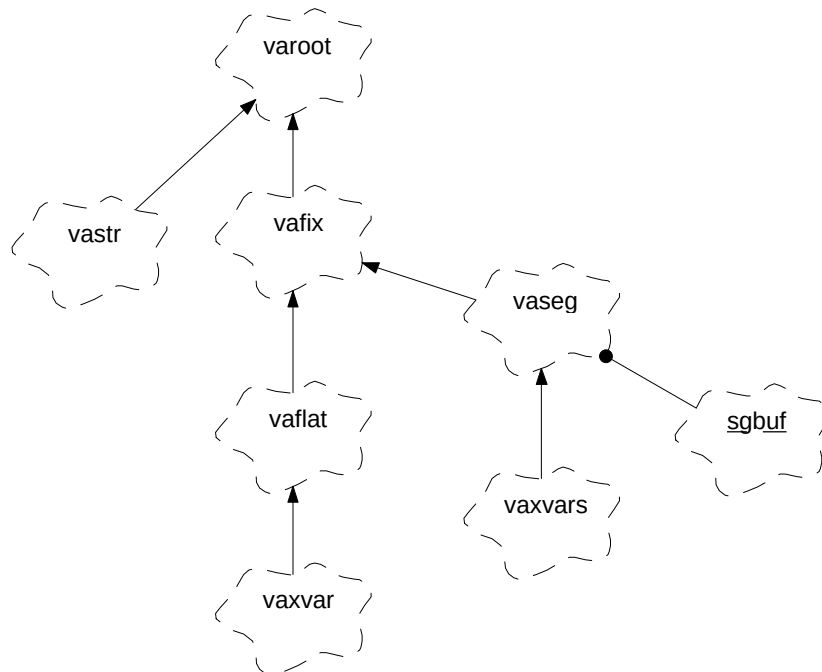
Extending the capacity of an array generally involves either allocating additional heap cells or growing a heap cell using `p_realloc`. This is always done in such a way that the original handle to the object does not change.

Precursors

An understanding of the variable array classes will be aided by a knowledge of:

- the PLIB memory allocator functions
- the `p_enter` and `p_leave` error handling services

Class diagram



Usage summary

`VAROOT` and `VAFIX` are abstract classes. `VAROOT` defines a relatively large number of deferred methods in order to promote polymorphism between the directly usable classes.

The `VASTR` class is used to create arrays of variable length text records, stored as zero terminated strings. The storage overhead per string is only one byte, so it is particularly suitable for storing short strings, of up to, say, several tens of bytes, or for strings with a wide variation in size (such as file names, which can be any length up to 128 bytes, but are usually much shorter). Since the whole array is stored in a single allocated cell, the `VASTR` class is most suitable for arrays that contain:

- a small number of records
- a moderately large, but fixed maximum, number of records (for which the maximum capacity can be allocated in advance)

Because of its suitability for storing file names, the `VASTR` class is used, for example, to store directory listings. In general, however, the `VASTR` class is not particularly suitable for arrays which can dynamically grow to a very large size. The resulting repeated calls to `p_realloc` are likely to cause heap fragmentation and seriously reduce the effective use of memory.

The `VAFLAT` class is used to create arrays of fixed length records, where the whole array is stored in a single allocated cell. The preferred usage is as for the `VASTR` class.

The `VASEG` class is used to create arrays of fixed length records where the array is segmented into a number of equal sized blocks. It is suitable for large, dynamically changing arrays. Its disadvantage is that it takes longer to locate a random record by record number because it has to count through the segments, although sequential access to the records is reasonably efficient.

The `VAXVAR` class is used to create arrays of variable length records which, unlike those of the `VASTR` class, may contain arbitrary data. It uses an index which is stored in a single allocated cell and thus, like `VASTR` and `VAFLAT`, is best suited to arrays containing either a small number of records or a larger but fixed number of records. Since each record is stored in a separate allocated cell, it is more suited to the storage of longer records, where the increased overhead per record is less significant.

The `VAXVARS` class (which has a segmented index) should be used instead of `VAXVAR` when there is a possibility of growth to a large number of records.

Record pointers

Many of the variable array methods take a record pointer `prec` as a parameter. The `va_prec` method, defined as a deferred method by `VAROOT`, and which converts a record number into a `prec` record pointer, assumes that each record is stored in such a way that it is possible to provide a record pointer that is equivalent to an external pointer. In most subclasses this assumption is valid. Subclasses that are, because of their internal structure, unable to provide a `va_prec` method may still inherit usefully from `VAROOT`, but should replace all inherited methods which rely on `va_prec` (`va_findisq`, `va_search` and `va_compare`).

In most, but not all, subclasses of `VAROOT`, `prec` is the address of the record data. A more general interpretation of `prec` is that it is a handle to a record in the array. In a variable length record array subclass, for example, `prec` might be the address of a *string descriptor* which contains the address and length of a buffer containing the record.

In contrast with `va_prec`, the `va_pbuf` method converts a record number into a pointer that is guaranteed to point to the record data. Although in most cases (see, for example, `VAFLAT` and `VASTR`) `va_prec` and `va_pbuf` return identical pointers, they would return different addresses in the case mentioned in the previous paragraph (see also the `VAXVAR` and `VAXVARS` classes).

In general, there is no guarantee that any method will not cause the data of one or more records to move in memory. An application should therefore always access records by record number and should not store the pointers supplied by either the `va_prec` or the `va_pbuf` method.

When scanning records, the user should not rely on the assumption of contiguous storage to scan through the records. Although that is true for some types of variable array, it is certainly not true in general.

None of the supplied methods ever refers to more than two record pointers at any one time. This means that the data of a variable array may be stored in a separate segment and only the last two accessed records need to be copied into the local data space.

VAROOT

VAROOT	
nrec	key
destroy	va_replace
va_count	
va_delete	va_copy
va_sort	va_reclen
va_key	va_swap
va_findisq	va_init
va_insertisq	va_deletem
va_append	va_insertm
va_insert	va_prec
va_search	va_pbuf
va_compare	va_capacity
va_reset	va_compress
va_test	

The VAROOT class is an abstract class which must be subclassed to provide a usable variable array class.

This abstract class does not assume that the records are of fixed length. It is designed so that it may be subclassed by classes in which the records are of either fixed or variable length.

The assumption that prec is a pointer to the record itself is made by only one method in this class - va_test. If this assumption is not true for a particular subclass, that subclass should replace va_test with a more appropriate method.

Class definition

Defined in sub-category file *varray.cl* (generated header file *varray.g*).

```

CLASS varoot root
The root class for variable arrays.
{
  REPLACE destroy      Send itself a va_reset message first
  ADD va_count          Return record count
  ADD va_delete         Delete a record
  ADD va_sort           Sort array
  ADD va_key            Set key parameters
  ADD va_findisq        Find a record in an ordered array
  ADD va_insertisq      Insert a record in sequence
  ADD va_append         Append after last record
  ADD va_insert         Insert before a record
  ADD va_search         Search for a match
  ADD va_compare        Compare two records
  ADD va_reset          Reset to zero records and capacity
  ADD va_test           Compare a record with a test record
  ADD va_replace        Replace a record
  DEFER va_copy         Copy a record
  DEFER va_reclen       Return record length
  DEFER va_swap         Swap two records
  DEFER va_init         Initialise an array
  DEFER va_deletem      Delete a record range
  DEFER va_insertm      Insert a record sequence
  DEFER va_prec         Return the address of record n
  DEFER va_pbuf         Point at the buffer data - usually same as prec
  DEFER va_capacity     Set record capacity
  DEFER va_compress     Compress memory usage

  CONSTANTS
  {
    VA_ROOT_DUPLICATE 1
    VA_ROOT_FLG_FOLD  0x01
    VA_ROOT_FLG_DESC  0x02
  }
}

```

```
TYPES
{
    typedef struct
    {
        UBYTE ofs;        offset for comparison
        UBYTE len;        length for bcmp (scmp if zero)
        UBYTE fold;       fold case if set
        UBYTE desc;       reverse compare result if set
    } PR_VARROOT_KEY;
}

PROPERTY
{
    UWORD nrec;          number of records in the array
    PR_VARROOT_KEY key;  defines key for sort etc
}
}
```

Property

<code>varroot.nrec</code>	Holds the current record count. Each subclass is expected to maintain this field.
<code>varroot.key</code>	Holds the current sort key. See the <code>va_key</code> method for a description of the <code>PR_VARROOT_KEY</code> structure fields.

VAROOT methods

★ DESTROY Destroy

`VOID destroy(VOID)`

Destroy the array by sending itself a `VA_RESET` message before supersending a `DESTROY` message.

★ VA_COUNT Count records

`UINT va_count(VOID)`

Return the number of records in the array from `varroot.nrec`.

VA_APPEND Append a record

`VOID va_append(VOID *prec)`

Append the record pointed to by `prec` to the end of the array.

This is done by sending itself a `VA_INSERT` message to insert the record at a position given by `varroot.nrec`.

It will call `p_leave` if the `va_insert` method for that particular subclass calls `p_leave`, for example, if it fails to allocate any necessary additional memory.

VA_INSERT Insert a record

`VOID va_insert(UINT recno, VOID *prec);`

Insert the record pointed to by `prec` before record `recno`.

This is done by sending itself a `VA_INSERTM` message with `recno`, `prec` and 1 (i.e. one record) as parameters.

It will call `p_leave` if the `va_insertm` method for that particular subclass calls `p_leave`, for example, if it fails to allocate any necessary additional memory.

★ VA_DELETE**Delete a record**

```
VOID va_delete(UINT recno);
```

Delete record `recno` from the array by sending itself a `VA_DELETEM` message with `recno` and 1 (i.e. one record) as parameters.

★ VA_KEY**Set key for comparisons**

```
VOID va_key(UINT offset, UINT length, UINT flags);
```

Define the parameters which are used by `va_test` and, indirectly, by `va_compare`, `va_sort`, `va_findisq` and `va_insertisq`.

The parameters are:

`offset` - sets the offset (0 to 127 inclusive) into the record at which the comparison begins. The value is copied into `varoot.key ofs`.

`length` - if non-zero, this sets the length of the comparison (1 to 127 inclusive) and, if zero, sets the comparison to be between two zero terminated strings. The value is copied into `varoot.key len`.

`flags` - any combination of the two flags `VA_ROOT_FLG_FOLD` and `VA_ROOT_FLG_DESC`. If the `VA_ROOT_FLG_FOLD` flag is set, the comparison is case independent. If the `VA_ROOT_FLG_DESC` flag is set, the result of the comparison is reversed, to give descending rather than ascending order. The values of `flags&VA_ROOT_FLG_FOLD` and `flags&VA_ROOT_FLG_DESC` are copied into `varoot.key.fold` and `varoot.key.desc` respectively.

The default settings are `offset = 0`, `length = 0`, `flags = 0`, giving ascending order, case-dependent string comparisons, from offset zero in the record buffer.

See the `VA_TEST` method for further discussion of the `VA_KEY` parameters.

★ VA_TEST**Compare two records by pointer**

```
INT va_test(VOID *prec1, VOID *prec2);
```

Compare the record pointed to by `prec2` with the record at `prec1` on the assumption that the records contain text. The basis for the comparison is subject to the contents of `varoot.key`, as set by the method `va_key`, as follows:

if `varoot.key.len==0` it uses
 `p_scmp` (if `varoot.key.fold` is FALSE), or
 `p_scmpi` (if `varoot.key.fold` is TRUE).

if `varoot.key.len>0` it uses
 `p_bcmp` (if `varoot.key.fold` is FALSE), or
 `p_bcmpi` (if `varoot.key.fold` is TRUE).

The default is to use `p_scmp`.

Returns the logical equivalent of `(*prec1 - *prec2)` - that is, zero if the two records are equal, negative if `*prec1` is before (less than) `*prec2`, positive if after. If `key.desc` is TRUE this result is reversed.

This method is used directly by `va_findisq`, `va_search` and `va_compare`. It is used indirectly by `va_sort` and `va_insertisq`.

Subclasses which are unable to provide `va_prec`, or in which `va_prec` does not return a pointer to the data to be compared, must replace this method.

★ VA_COMPARE**Compare two records by number**

```
INT va_compare(UINT n1, UINT n2);
```

Compare record `n1` with record `n2` by sending itself `VA_PREC` messages to get pointers to the records and then sending itself a `VA_TEST` message to perform the comparison that determines the return value.

Returns the logical equivalent of `n1 - n2`, that is, zero if the two records are equal, a negative value if record `n1` is less than (before) record `n2` or a positive value if record `n1` is greater than (after) record `n2`. Note the effect of the `VA_ROOT_FLG_FOLD` and `VA_ROOT_FLG_DESC` flags.

This method is used directly by `va_sort`.

Subclasses which are unable to provide `va_prec` must replace this method.

VA_SORT**Sort**

```
VOID va_sort(VOID)
```

Sort the records of the array.

The sort uses the quicksort algorithm. This is an efficient exchange sort using `va_compare` and the deferred `va_swap` respectively to compare and to exchange two records.

Subclasses which are unable to implement `va_swap` efficiently should either not support `va_swap` or subclass `va_sort` to sort by some other means. Not supporting `va_swap` does not mean that the array may never be ordered; ordered arrays can still be constructed using `va_insertisq`.

The method uses `va_compare` (and hence `va_test`) to compare two records. The sort key is determined by `va_test`, as qualified by the last use of `va_key`. If the flexibility afforded by `va_key` is insufficient (for example, to sort on multiple keys) `va_test` may be replaced.

Note that the `VASTR` class does not support this method. In such a case the array may be built in order, by using the `va_insertisq` method.

★ VA_FINDISQ**Find (binary chop)**

```
INT va_findisq(VOID *pkey, VOID *pmid);
```

Find a record in an ordered array, using the binary search algorithm.

The record to be found is specified by `pkey` which is typically a record pointer (`prec`).

Returns zero if an exact record match is found, with the record number of the matching record in `*pmid`.

If there is no matching record, `*pmid` contains the record number of one of the two records adjacent to the key and `va_findisq` returns the logical equivalent of `*pkey - *pmid`, that is, a negative value if `*pkey` is less than (before) the existing record with record number `*pmid`, or a positive value if `*pkey` is greater than (after) record number `*pmid`.

The method uses `va_test`, passing `pkey` as the first parameter; the second is a pointer to one of the records in the ordered variable array and is determined by the binary search algorithm itself as it works through its search.

The result will be unpredictable if the array is not ordered. (An array may be ordered either by applying `va_sort` or by using `va_insertisq` to build the array.)

VA_INSERTISQ**Insert in sequence**

```
INT va_insertisq(VOID *prec, UWORD *precno);
```

Insert the record pointed to by `prec` in sequence into an ordered array using `va_findisq` to locate the insertion point.

If there is no matching record, it sends itself a `VA_INSERTM` message. If the insertion is successful, the record number of the inserted record is written to `*precno` and `va_insertisq` returns zero.

The record is not inserted if there is already a matching record in the array (that is, if `va_compare` would return zero). In this case `va_insertisq` returns `VA_ROOT_DUPLICATE` (which is a positive number) and writes the matching record number to `*precno`.

Since it sends a `VA_INSERTM` message, it is quite possible for the insert to fail with out of memory and call `p_leave`.

★ VA_SEARCH**Search**

```
INT va_search(VOID *pkey);
```

Sequentially search for a record which *exactly* matches that specified by `pkey`.

The search is performed by sending a `VA_TEST` message to compare each record in the array (obtained by use of `va_prec`) with the data pointed to by `pkey`, which is assumed to be a record pointer.

Returns the record number if found (+ve number or zero) or `E_GEN_FAIL` if not.

★ VA_RESET**Reset**

```
VOID va_reset(VOID)
```

Reset the array to its state just after its creation (and, if appropriate, a `va_init`). Following a `va_reset` the array contains no records and has no record capacity.

The method is implemented by sending itself a `VA_DELETEM` message to delete `varoot.nrec` records, starting from record 0. This is followed by a `VA_COMPRESS` message to discard all record capacity.

The method has two principal uses:

- It provides the client with a concise and efficient way to delete all records in the array and, at the same time, to zero the capacity.
- it is used by the `destroy` method to remove all allocated cells, other than the object instance cell, prior to the freeing of the instance itself.

This implementation assumes that the `va_compress` method always frees all additional allocated cells when an array contains no records. (This assumption is true for all OLIB array classes.)

VA_REPLACE**Replace a record**

```
VOID va_replace(UINT recno, VOID *prec);
```

Replace record number `recno` with the record pointed to by `prec`.

The method is implemented by using a `VA_DELETE` message to delete record `recno`, followed by a `VA_INSERT` message to insert `prec` at position `recno`.

The implementation of this method is aimed at variable length record subclasses; fixed length record subclasses can replace it by a more efficient method (for example, by simply overwriting the record data).

Deferred VAROOT methods**★ VA_COPY****Copy a record**

```
UINT va_copy(UINT recno, VOID *prec);
```

A deferred method for copying the contents of record `recno` to `prec`.

Returns the length copied.

Not used by any methods in this class but deferred, to promote polymorphic subclasses.

★ VA_RECLN**Get record length**

```
UINT va_recln(UINT recno);
```

A deferred method for returning the record length of record `recno`.

Not used by any methods in this class but deferred, to promote polymorphic subclasses.

VA_SWAP**Swap two records**

```
VOID va_swap(UINT n1, UINT n2);
```

A deferred method for swapping the contents of record `n1` with those of record `n2`.

Used by `va_sort`.

VA_INIT**Initialise**

```
VOID va_init(...)
```

A deferred method for initialising the property of the newly created array.

No record capacity is allocated until the first record insertion or `VA_CAPACITY` message. All subclasses should ensure that this is the case. However, the method may still fail, calling `p_leave(E_GEN_NOMEMORY)` if the subclass has other memory requirements (`VASEG`, for example, creates a component object).

The parameters depend upon the subclass. For example, a fixed length record subclass would require the record length as a parameter.

Not used by any methods in this class but deferred, to promote polymorphic subclasses.

VA_CAPACITY**Set capacity**

```
VOID va_capacity(UINT nspc);
```

A deferred method for setting the capacity of the internal storage.

The interpretation of the parameter `nspc` is dependent on the subclass. For example, `nspc` might reasonably be the number of records in a fixed length record subclass, or it might be the total number of bytes of allocated storage in a variable length record subclass.

It is not always possible to provide this method in a meaningful way and some subclasses are expected to dummy it.

Not used by any methods in this class but deferred, to promote polymorphic subclasses.

VA_COMPRESS**Compress**

```
VOID va_compress(VOID);
```

A deferred method for compressing the capacity of the array as much as is reasonable and sensible (this judgement is left to the subclass).

The supplied `destroy` method assumes that `va_compress` frees all allocated cells used to hold records when there are no records in the array. If this is not true, the `destroy` method must be subclassed.

The `va_compress` method is used directly by `va_reset` and indirectly by `destroy`.

VA_DELETEM**Delete sequence of records**

```
VOID va_deletem(UINT recno, UINT nrecs);
```

A deferred method for deleting `nrecs` records, starting with record number `recno`.

Used directly by `va_delete` and `va_reset` and indirectly by `destroy`.

VA_INSERTM**Insert sequence of records**

```
VOID va_insertm(UINT recno, VOID *prec, UINT nrecs);
```

A deferred method for inserting a sequence of `nrecs` records, pointed to by `prec`, before record `recno`.

If it fails to allocate enough memory to hold the new records it should allocate nothing and call `p_leave` since, typically, `va_insert` and `va_append` are void functions and `va_insertisq` returns an insertion indicator.

Used directly by `va_insert` and indirectly by `va_append` and `va_insertisq`.

VA_PREC**Point to record**

```
VOID *va_prec(UINT recno);
```

A deferred method for returning a pointer to record *recno*.

See the introductory discussion of the `VAROOT` class for the meaning of a record pointer.

Used directly by `va_compare`, `va_findisq` and `va_search`, and indirectly by `va_insertisq` and `va_sort`.

VA_PBUF**Point to record data**

```
VOID *va_pbuf(UINT recno);
```

A deferred method for returning a pointer to the record *data* of record *recno*.

Typically, it returns the same value as for `va_prec`. in some classes, however, the record pointer may not be the same as the record data pointer (this is true for the `VAXVAR` and `VAXVARS` classes) or the record may contain a header to the data.

The `va_pbuf` method should always be used in preference to `va_prec` when a pointer to the data of the record is required.

VAFIX

VAROOT		VAFIX
nrec	key	r len
destroy	~~va_replace~~	va_replace
va_count		va_copy
va_delete	~~va_copy~~	va_reclen
va_sort	~~va_reclen~~	va_swap
va_key	~~va_swap~~	
va_findisq	va_init	
va_insertisq	va_deletem	
va_append	va_insertm	
va_insert	va_prec	
va_search	va_pbuf	
va_compare	va_capacity	
va_reset	va_compress	
va_test		

The `VAFIX` class is an abstract class for fixed length record variable arrays. In addition to adding the record length `r len` to the property, it:

- implements the deferred methods `va_swap`, `va_copy` and `va_reclen`.
- replaces `va_replace` by a more efficient method.

The methods are provided on the assumption that the record pointer `prec` is simply the address of the record (reasonable when records are of fixed length).

Class definition

Defined in sub-category file *varray.cl* (generated header file *varray.g*).

```
CLASS vafix varoot
Fixed length record variable arrays.
{
  REPLACE va_replace      More efficient than varoot's
  REPLACE va_swap         Uses p_bswap
  REPLACE va_copy         Uses p_bcpy
  REPLACE va_reclen       Returns property value
  PROPERTY
  {
    UWORD rlen;           Record length
  }
}
```

Property

`vafix.rlen` the record length, set by a subclass `va_init` method. Each subclass is expected to set this field.

VAFIX methods

★ VA_REPLACE

Replace a record

```
VOID va_replace(UINT recno, VOID *prec);
```

Replace the specified record by sending itself a `VA_PREC` message to convert `recno` into a record pointer and then using `p_bcpy` to overwrite that record with the record at `prec`.

★ VA_COPY

Copy a record

```
UINT va_copy(UINT recno, VOID *prec);
```

Copy the specified record to `prec` by sending itself a `VA_PREC` message to convert `recno` into a record pointer and then using `p_bcpy` to copy that record data from the array to `prec`.

Returns the length copied.

★ VA_RECLEN

Get record length

```
UINT va_reclen(VOID)
```

Return the record length, that is, the value of the `vafix.rlen` property field.

★ VA_SWAP

Swap two records

```
VOID va_swap(UINT n1,UINT n2)
```

Swap the contents of record `n1` with record `n2` by sending two `VA_PREC` messages to itself to turn `n1` and `n2` into record pointers and then using `p_bswap` to swap the record contents.

VASTR

VAROOT	
nrec	key
destroy	va_replace
va_count	
va_delete	~~va_copy~~
va_sort	~~va_reclen~~
va_key	va_swap
va_findisq	~~va_init~~
va_insertisq	~~va_deletem~~
va_append	~~va_insertm~~
va_insert	~~va_prec~~
va_search	~~va_pbuf~~
va_compare	~~va_capacity~~
va_reset	~~va_compress~~
va_test	

VASTR	
size	
gran	
base	
len	
num	
pnum	
va_copy	
va_reclen	
va_init	
va_deletem	
va_insertm	
va_prec	
va_pbuf	
va_capacity	
va_compress	

The VASTR class may be used to create arrays of variable length text records which are stored as zero terminated strings. Non-textual data may be used provided that the record data does not have any zero bytes in it.

The whole array is stored in a single allocated cell. When a record is inserted into a full array, the single cell is reallocated to accommodate the additional record. Deletions do not automatically reduce the record capacity, but the capacity may be reduced manually using `va_compress` or `va_capacity`.

The record pointer `prec` points to a zero terminated string. Internally, the records are stored as a contiguous sequence of zero terminated strings.

To locate a random record by record number, `va_prec` has to count through the records. However, the object remembers the last record accessed so that scanning the array sequentially from the first record is reasonably efficient.

The VASTR class is suitable for short arrays or for large arrays which have a known maximum capacity (in terms of the number of bytes required). It is not suitable for arrays which can dynamically grow to a large size because heap fragmentation can seriously reduce the effective use of the heap.

The string array is especially suitable for strings that can vary greatly in size since the overhead per string (1 byte) is small. A typical use is for holding file name lists where each string can have a maximum length of `P_FNAME_SIZE` (128) bytes but is usually less than 32 bytes long.

Class definition

Defined in sub-category file *varray.cl* (generated header file *varray.g*).

```
CLASS   vastr   varoot
Variable length text record variable arrays
{
  REPLACE va_copy
  REPLACE va_reclen
  REPLACE va_init
  REPLACE va_compress
  REPLACE va_capacity
  REPLACE va_deletem
  REPLACE va_insertm
  REPLACE va_prec
  REPLACE va_pbuf=vastr_va_prec
  PROPERTY
  {
    UWORD size;          current size of the array in bytes
    UWORD gran;          re-alloc granularity
    UBYTE *base;         base of the array
    UWORD len;           offset to end of used data
    UWORD num;           number of last record referenced
    UBYTE *pnum;         pointer to record num
  }
}
```

Property

<code>vastr.size</code>	The current size of the allocated cell that contains the data. It should not be accessed by any subclass.
<code>vastr.gran</code>	The granularity, <i>in bytes</i> , used when expanding the allocated cell. It should not be accessed by any subclass.
<code>vastr.base</code>	The allocated cell base. It should not be accessed by any subclass.
<code>vastr.len</code>	The byte offset from <code>vastr.base</code> to the end of the data in the allocated cell. It should not be accessed by any subclass.
<code>vastr.num</code>	The record number of last record referenced. It should not be accessed by any subclass.
<code>vastr.pnum</code>	A pointer to the record specified by <code>vastr.num</code> . It should not be accessed by any subclass.

VASTR methods

★ VA_INIT

Initialise

```
VOID va_init(UINT granularity)
```

Initialise the array by setting `vastr.gran` to the passed `granularity`, which must be at least as large as the longest record to be inserted. No capacity is actually allocated until the first insertion.

The granularity is significant when an insertion requires an increase in capacity; the increase is such that the capacity, in bytes, is made an exact multiple of the granularity. Making this value larger means that the array cell needs to be reallocated less frequently as a result of insertions, but more memory may be wasted in unused capacity. Any unused capacity may be recovered by sending a `VA_COMPRESS` message when the building of an array is complete.

VA_CAPACITY**Set capacity**

```
VOID va_capacity(UINT nspc);
```

Set the capacity of the allocated array cell by reallocating it to have exactly the capacity for `nspc` bytes (the granularity has no effect).

The minimum space required is equal to the sum of the string lengths of all the records plus one byte per record for the terminating zero.

Calls `p_panic(P_PANIC_P_VASTR_2)` if `nspc` is less than the current size of the array.

★ VA_COMPRESS**Compress**

```
VOID va_compress(VOID)
```

Compress the capacity of the array to that which will exactly contain the current records by sending itself a `VA_CAPACITY` message with `vastr.len` as the size.

If there are no records in the array, the array cell is freed (as required by `VAROOT`).

★ VA_DELETEM**Delete a sequence of records**

```
VOID va_deletem(UINT num, UINT nrecs);
```

Delete the sequence of `nrecs` records, starting at record number `num`, and decrease the value of `varoot.nrec` by `nrecs`.

The deletion is performed by simply copying all following records over the records to be deleted. No memory is freed.

Calls `p_panic(P_PANIC_P_VASTR_4)` if `num+nrecs` is greater than the number of records in the array.

VA_INSERTM**Insert a sequence of records**

```
VOID va_insertm(UINT num, VOID *prec, UINT nrecs);
```

Insert the sequence of `nrecs` records, pointed to by `prec`, before record number `num`, and increase the value of `varoot.nrec` by `nrecs`.

The valid range for `num` is from zero to the number of records inclusive.

In the record sequence at `prec`, each subsequent string should immediately follow the zero terminator of the previous string.

Calls `p_panic(P_PANIC_P_VASTR_3)` if `num` is greater than the number of records in the array.

Inserts no records and calls `p_leave(E_GEN_NOMEMORY)` if there is not enough memory available to hold all the additional records.

★ VA_RECLEM**Get record length**

```
UINT va_reclen(UINT num);
```

Return the record length of record number `num`. The returned length excludes the zero terminator.

★ VA_PREC**Point to record**

```
VOID *va_prec(UINT num);
```

Return the address of record number `num`. The pointer returned is to a zero terminated string.

Calls `p_panic(P_PANIC_P_VASTR_1)` if `num` is greater than or equal to the number of records in the array.

★ VA_PBUF

Point to record data

VOID *va_pbuf(UINT num)

As VA_PREC.

★ VA_COPY

Copy a record

UINT va_copy(UINT num, VOID *prec);

Copy the record specified by the parameter num to the location pointed to by the parameter prec; by sending itself a VA_PREC message to convert recno into a record pointer and then using p_bcopy to copy that record data from the array to prec.

Returns the length copied.

VAFLAT

VAROOT		VAFIX	VAFLAT
nrec	key	rlen	gran nspc base
destroy	~~va_replace~~	va_replace	va_init
va_count		va_copy	va_compress
va_delete	~~va_copy~~	va_reclen	va_deletem
va_sort	~~va_reclen~~	va_swap	va_insertm
va_key	~~va_swap~~		va_capacity
va_findisq	~~va_init~~		va_prec
va_insertisq	~~va_deletem~~		va_pbuf
va_append	~~va_insertm~~		
va_insert	~~va_prec~~		
va_search	~~va_pbuf~~		
va_compare	~~va_capacity~~		
va_reset	~~va_compress~~		
va_test			

The VAFLAT class may be used to create arrays of fixed length records where the whole array is stored in a single allocated cell. When a record is inserted into a full array, the single cell is reallocated to accommodate the additional record. Deletions do not automatically reduce the record capacity but the capacity may be reduced manually using va_compress or va_capacity.

The VAFLAT class is suitable for short arrays or for large arrays which have a known maximum capacity. It is not suitable for arrays which can dynamically grow to a large size because heap fragmentation can seriously reduce the effective use of heap memory.

Class definition

Defined in sub-category file *varray.cl* (generated header file *varray.g*).

```

CLASS vaflat vafix
Flat allocated (in a single cell) fixed length variable arrays
{
  REPLACE va_init
  REPLACE va_compress
  REPLACE va_deletem
  REPLACE va_insertm
  REPLACE va_capacity
  REPLACE va_prec
  REPLACE va_pbuf=vaflat_va_prec
  PROPERTY
  {
    UWORD gran;    granularity in records
    UWORD nspc;    present record capacity
    UBYTE *base;   start of variable array
  }
}

```

Property

<code>vaflat.gran</code>	The granularity, <i>in records</i> , in which to allocate memory. It should not be accessed by any subclass.
<code>vaflat.nspc</code>	The number of fixed sized record slots allocated (greater than or equal to the number of records in the array). It should not be accessed by any subclass.
<code>vaflat.base</code>	The allocated space handle, i.e. the start of the variable array. It should not be accessed by any subclass.

VAFLAT methods**★ VA_INIT****Initialise**

```
VOID va_init(UINT reclen, UINT gran);
```

Initialise the array by setting `vafix.rlen` from `reclen` and `vaflat.gran` from `gran`. No capacity is actually allocated until the first insertion, hence no errors can occur.

The granularity is significant when an insertion requires an increase in capacity; the increase is such that the capacity, in records, is made an exact multiple of the granularity. Increasing the value of `gran` means that the array cell needs to be reallocated less frequently (hence less heap fragmentation) as a result of insertions, but more memory may be wasted in unused capacity. Any unused capacity may be recovered by sending a `VA_COMPRESS` message when the building of an array is complete.

VA_CAPACITY**Set capacity**

```
VOID va_capacity(UINT nspc);
```

Set the capacity of the array cell by reallocating it to have exactly the capacity for `nspc` records (i.e. `nspc*vafix.rlen` bytes).

Does not alter the capacity and calls `p_leave(E_GEN_NOMEMORY)` if there is not enough memory for the specified capacity.

Calls `p_panic(P_PANIC_P_VAFLAT_3)` if `nspc` is less than the current number of records in the array.

★ VA_COMPRESS**Compress**

```
VOID va_compress(VOID)
```

Compress the capacity of the array to exactly that required to hold the current records. This is achieved by sending itself a `VA_CAPACITY` message to set the capacity to `varoot.nrec` records. If there are no records in the array, the array cell is freed as required by the `VAROOT` class.

★ VA_DELETEM**Delete sequence of records**

```
VOID va_deletem(UINT recno, UINT nrecs);
```

Delete the sequence of `nrecs` records starting at record `recno`.

Calls `p_panic(P_PANIC_P_VAFLAT_4)` if `recno+nrecs` is greater than the number of records in the array.

The number of records in the array, held in `varoot.nrec`, is reduced by `nrecs`. The delete is performed by copying the data of following records over the records being deleted. No allocated space is freed.

VA_INSERTM**Insert sequence of records**

```
VOID va_insertm(UINT recno, VOID *prec, UINT nrecs)
```

Inserts the sequence of `nrecs` records pointed to by `prec` before record `recno` where `recno` is between zero and the number of records inclusive.

If the array cannot currently hold all the additional records, the record capacity is set (by sending itself a `VA_CAPACITY` message) to the smallest exact multiple of the granularity that is greater than the total number of records to be held. The data is inserted by opening up a gap in the array (by buffer copying) then copying the new data into the gap.

Inserts no records and calls `p_leave(E_GEN_NOMEMORY)` if there is not enough memory available to hold the additional records.

Calls `p_panic(P_PANIC_P_VAFLAT_2)` if `recno` is greater than the number of records in the array.

★ VA_PREC

Point to record

```
VOID *va_prec(UINT recno)
```

Return the address of record `recno`.

Calls `p_panic(P_PANIC_P_VAFLAT_1)` if `recno` is greater than or equal to the number of records in the array.

★ VA_PBUF

Point to record data

```
VOID *va_pbuf(UINT recno);
```

As `VA_PREC`.

VASEG

VAROOT		VAFIX	VASEG
nrec	key	rlen	buf
destroy	~~va_replace~~	va_replace	va_init
va_count		va_copy	va_compress
va_delete	~~va_copy~~	va_reclen	va_deletem
va_sort	~~va_reclen~~	va_swap	va_insertm
va_key	~~va_swap~~		va_capacity
va_findisq	~~va_init~~		va_prec
va_insertisq	~~va_deletem~~		va_pbuf
va_append	~~va_insertm~~		
va_insert	~~va_prec~~		
va_search	~~va_pbuf~~		
va_compare	~~va_capacity~~		
va_reset	~~va_compress~~		
va_test			

The `VASEG` class may be used to create arrays of fixed length records where the array is segmented into a number of equal sized blocks. The segments are allocated from the heap and are doubly linked.

The segmented array object is suitable for large dynamically changing arrays and is substantially more likely to make efficient use of the available heap memory. Its disadvantage is that it takes longer to locate a random record by record number because it has to count through the segments. However, the object remembers the last record accessed so that scanning a segmented array sequentially from the first record is reasonably efficient.

The segments are generally not less than 50% full. See the `SGBUF` class documentation (in the *SGBUF Segmented Buffer Class* chapter) for more information on the segmentation mechanisms.

Class definition

Defined in sub-category file *varray.cl* (generated header file *varray.g*).

```

CLASS   vaseg   vafix
Segmented fixed length variable arrays
{
  REPLACE va_init
  REPLACE va_compress
  REPLACE va_capacity=p_dummy
  REPLACE va_deletem
  REPLACE va_insertm
  REPLACE va_prec
  REPLACE va_pbuf=vaseg_va_prec
  PROPERTY 1
    {
      PR_SGBUF *buf;      segmented buffer
    }
}

```

Property

`vaseg.buf` The object handle of the owned instance of the segmented buffer (SGBUF) class. This may be used by any subclass to access the SGBUF object.

VASEG methods
VA_INIT**Initialise**

`VOID va_init(UINT rlen, UINT granularity);`

Initialise the array by setting the record length to `rlen`, creating a SGBUF object and initialising it with a segment size of `rlen*granularity`. Note that, since the segment size is an exact multiple of the record length, the content of a record will never straddle a segment boundary (significant for the `va_prec` and `va_pbuf` methods).

The first segment is not allocated until the first insertion.

Calls `p_leave(E_GEN_NOMEMORY)` if it cannot create the SGBUF object.

VA_CAPACITY**Set capacity**

Setting the capacity does not make any sense for the `vaseg` class and this method does nothing.

★ VA_COMPRESS**Compress**

`VOID va_compress(VOID)`

Compress the capacity of the array by sending an `SB_COMPRESS` message to the owned SGBUF object.

★ VA_DELETEM**Delete a sequence of records**

`VOID va_deletem(UINT recno, UINT nrecs);`

Delete the sequence of `nrecs` records starting at record `recno`. The delete is performed by sending an `SB_DELETE` message to the owned SGBUF object.

Reduces the record count by `nrecs`.

VA_INSERTM Insert a sequence of records

```
VOID va_insertm(UINT recno, VOID *prec, UINT nrecs);
```

Insert the sequence of `nrecs` records pointed to by `prec` before record `recno`, where `recno` is between zero and the number of records inclusive. The insert is performed by sending an `SB_INSERT` message to the owned `SGBUF` object.

Increases the record count by `nrecs` if the insertion was successful.

Inserts no records and calls `p_leave(E_GEN_NOMEMORY)` if there is not enough memory available to hold the additional records.

★ VA_PREC Point to record

```
VOID *va_prec(UINT recno)
```

Returns the address of record number `recno`.

The record pointer is obtained by sending an `SB_POINT` message to the owned `SGBUF` object.

Calls `p_panic(P_PANIC_P_VASEG_1)` if `recno` is greater than or equal to the number of records in the array.

★ VA_PBUF Point to record data

```
VOID *va_pbuf(UINT recno)
```

As `va_prec`.

VAXVAR

VAROOT		VAFIX	VAFLAT	VAXVAR
nrec	key	rlen	gran nspc base	
destroy	~~va_replace~~	~~va_replace~~	~~va_init~~	va_test
va_count		~~va_copy~~	va_compress	va_copy
va_delete	~~va_copy~~	~~va_reclen~~	~~va_deletem~~	va_reclen
va_sort	~~va_reclen~~	va_swap	~~va_insertm~~	va_replace
va_key	~~va_swap~~		va_capacity	va_init
va_findisq	~~va_init~~		va_prec	va_deletem
va_insertisq	~~va_deletem~~		~~va_pbuf~~	va_insertm
va_append	~~va_insertm~~			va_pbuf
va_insert	~~va_prec~~			
va_search	~~va_pbuf~~			
va_compare	~~va_capacity~~			
va_reset	~~va_compress~~			
va_test				

The `VAXVAR` class may be used to create arrays of variable length records where there is no restriction on the values of the bytes which can be stored within records. Each record is stored in its own heap cell and the records are indexed by a `VAFLAT` array of `RC_VAXVAR` structs.

The `VAXVAR` class subclasses the fixed length record array `VAFLAT` which provides its index. Although `VAXVAR` has variable length records, only 8 of the methods inherited from `VAFLAT` needed to be replaced.

In `VAXVAR`, records are not described simply by their address but indirectly via the address of a record descriptor. A record descriptor is a `RC_VAXVAR` struct that contains the address and length of a record.

The `RC_VAXVAR` struct is used to specify records both outside and inside the array. For example, the insertion methods `va_append`, `va_insert`, `va_insertisq` and `va_insertm` all require the address of an `RC_VAXVAR` struct - and the `va_prec` method returns the address of an `RC_VAXVAR` struct.

The `va_pbuf` method will return a pointer to the actual data buffer (ie the `RC_VAXVAR buf` field). The `va_copy` method takes a buffer address (to take the copied record) rather than the address of an `RC_VAXVAR` struct.

When a record is inserted, a cell is allocated for the variable length record data and an index record is inserted - the capacity of the index may need to be increased. When a record is deleted, the record data cell is freed and the corresponding index record is deleted. Deletions do not automatically reduce the record capacity of the index but the index capacity may be reduced manually using `va_compress` or `va_capacity`. There is no means of controlling the capacity of the record data storage.

On a 16-bit address machine, the overhead per record is 4 bytes for the `RC_VAXVAR` record and at least 2 bytes for the allocated cell. However, any zero length records do not have an associated record value cell and, in this case, the corresponding `buf` (address) field of the `RC_VAXVAR` struct is guaranteed to be `NULL`.

The `VAXVAR` class is suitable for short to medium length arrays or for large arrays which have a known maximum capacity (in terms of the number of records required). It is not suitable for arrays which can dynamically grow to a large number of records because the index is in a single cell. The `VAXVARS` class (which has a segmented index) should be used when there is a possibility of growth to a large number of records.

Compared to `VASTR`, `VAXVAR` has the following advantages:

- `VAXVAR` can store arbitrary record data, which may include zero bytes
- random access to `VAXVAR` records is efficient
- `VAXVAR` records may be exchanged efficiently since only the corresponding index items are exchanged - the sort method is thus very efficient
- the data for each record is stored in a separately allocated cell, so heap fragmentation is less likely to be a problem

Compared to `VASTR`, `VAXVAR` has the following disadvantages:

- the record overhead in `VAXVAR` is at least six bytes compared to one byte in `VASTR`
- the `RC_VAXVAR` descriptor which is used to describe a record is often less convenient than the address of a zero terminated string, but the `va_pbuf` method overcomes this quite well
- `VAXVAR` records do not automatically provide a zero terminator

Class definition

Defined in sub-category file `varray.cl` (generated header file `varray.g`).

```

CLASS vaxvar vaflat
Indexed variable length record variable arrays - one record per heap cell
{
  REPLACE va_test
  REPLACE va_copy
  REPLACE va_reclen
  REPLACE va_replace
  REPLACE va_init
  REPLACE va_deletem
  REPLACE va_insertm
  REPLACE va_pbuf
  TYPES
  {
    typedef struct
    {
      UWORD len;      record length
      UBYTE *buf;     record data
    } RC_VAXVAR;
  }
}

```

Property

None.

VAXVAR methods

★ VA_INIT

Initialise

```
VOID va_init(UINT gran);
```

Initialise the array by supersending the VA_INIT message to the VAFLAT object, passing the size of an RC_VAXVAR structure as the record size and gran as the granularity. The subclassed VAFLAT array provides the index used by the VAXVAR object. No capacity is actually allocated until the first insertion, hence no out of memory errors can occur.

The granularity is significant when an insertion requires an increase in capacity; the increase is such that the capacity, in records, is made an exact multiple of the granularity. Making this value larger means that the index array cell needs to be reallocated less frequently (hence less heap fragmentation) as a result of insertions, but more memory may be wasted in unused capacity.

★ VA_TEST

Compare two records by pointer

```
INT va_test(VOID *prec1, VOID *prec2);
```

Compare the record pointed to by prec2->buf with the record at prec1->buf, on the assumption that they contain text, exactly as for the VAROOT va_test method.

The basis for the comparison is subject to the contents of varoot.key, set by va_key, as follows:

```
if varoot.key.len==0 it uses
    p_scmp  (if varoot.key.fold is FALSE), or
    p_scmpi (if varoot.key.fold is TRUE).
```

```
if varoot.key.len>0 it uses
    p_bcmp  (if varoot.key.fold is FALSE), or
    p_bcmpi (if varoot.key.fold is TRUE).
```

The default is to use p_scmp.

Returns the logical equivalent of (*prec1->buf-*prec2->buf) - that is, zero if the two records are equal, negative if *prec1->buf is before (less than) *prec2->buf, positive if after. If key.desc is TRUE this result is reversed.

★ VA_DELETEM

Delete a sequence of records

```
VOID va_deletem(UINT recno, UINT nrecs);
```

Delete the sequence of nrecs records starting at record recno.

The process of deletion is two fold: first, the records defined by the RC_VAXVAR elements are found and any allocated cell (buffer data) is freed; Second, the index elements are deleted by supersending a VA_DELETEM message.

VA_INSERTM

Insert a sequence of records

```
VOID va_insertm(UINT recno, RC_VAXVAR *prec, UINT nrecs);
```

Insert the sequence of nrecs records pointed to by prec before record recno, where recno is between zero and the number of records inclusive. The record sequence pointed to by prec must be a contiguous array of nrecs RC_VAXVAR structs.

The insertion of records is a two stage process. Firstly the index entries are inserted by supersending a VA_INSERTM message to the VAFLAT superclass. If this is successful, a heap cell is allocated for each record to be inserted, the data is copied into each cell and the cell handle written to the RC_VAXVAR buf field. If an allocation fails, all entries made so far are removed, their allocated cells being freed.

Inserts no records and calls p_leave(E_GEN_NOMEMORY) if there is not enough memory available to hold all the additional records.

VA_REPLACE**Replace record**

```
VOID va_replace(UINT recno, RC_VAXVAR *prec);
```

Replace record `recno` with the record described by the `RC_VAXVAR` struct at `prec`, by freeing the original record cell, allocating a new cell of the appropriate size and replacing the index record with the new descriptor.

Calls `p_leave(E_GEN_NOMEMORY)`, without modifying the original record, if there is not enough memory available to hold the replacement record.

★ VA_COPY**Copy a record**

```
UINT va_copy(UINT recno, VOID *pbuf);
```

Copy record number `recno` by supersending itself a `VA_COPY` message to get the record descriptor and then using `p_bcopy` to copy the record data to `pbuf`.

Returns the length of the data copied to `pbuf`.

★ VA_RECLEN**Get record length**

```
UINT va_reclen(UINT recno);
```

Returns the record length of record `recno`.

★ VA_PBUF**Point to record data**

```
VOID *va_pbuf(UINT recno);
```

Returns a pointer to the record data, read from the `buf` field of the corresponding `RC_VAXVAR` struct.

It should be noted that the `va_prec` method (provided by the `VAFAT` superclass) returns a pointer to the `RC_VAXVAR` struct.

VAXVARS

VAROOT		VAFIX	VASEG	VAXVARS
<code>nrec</code>	<code>key</code>	<code>rilen</code>	<code>buf</code>	
<code>destroy</code>	<code>va_replace</code>	<code>va_replace</code>	<code>va_init</code>	<code>va_test</code>
<code>va_count</code>		<code>va_copy</code>	<code>va_compress</code>	<code>va_copy</code>
<code>va_delete</code>	<code>va_copy</code>	<code>va_reclen</code>	<code>va_deletem</code>	<code>va_reclen</code>
<code>va_sort</code>	<code>va_reclen</code>	<code>va_swap</code>	<code>va_insertm</code>	<code>va_replace</code>
<code>va_key</code>	<code>va_swap</code>		<code>va_capacity</code>	<code>va_init</code>
<code>va_findisq</code>	<code>va_init</code>		<code>va_prec</code>	<code>va_deletem</code>
<code>va_insertisq</code>	<code>va_deletem</code>		<code>va_pbuf</code>	<code>va_insertm</code>
<code>va_append</code>	<code>va_insertm</code>			<code>va_pbuf</code>
<code>va_insert</code>	<code>va_prec</code>			
<code>va_search</code>	<code>va_pbuf</code>			
<code>va_compare</code>	<code>va_capacity</code>			
<code>va_reset</code>	<code>va_compress</code>			
<code>va_test</code>				

The `VAXVARS` class is identical to the `VAXVAR` class except that it subclasses the `VASEG` class (a segmented array) for its index rather than `VAFAT` (a flat array). In other words, the index records are placed in a segmented buffer.

The `VAXVARS` class should be used in preference to `VAXVAR` when there is the possibility of a large number of records.

Class definition

Defined in sub-category file *varray.cl* (generated header file *varray.g*).

```
CLASS vaxvars vaseg
Indexed variable length record variable arrays - one record per heap cell
Uses a segmented array for an index
{
  REPLACE va_test=vaxvar_va_test
  REPLACE va_copy=vaxvar_va_copy
  REPLACE va_reclen=vaxvar_va_reclen
  REPLACE va_replace=vaxvar_va_replace
  REPLACE va_init=vaxvar_va_init
  REPLACE va_deletem=vaxvar_va_deletem
  REPLACE va_insertm=vaxvar_va_insertm
  REPLACE va_pbuf=vaxvar_va_pbuf
}
```

Property

None.

VAXVARS methods

See *VAXVAR methods*.

CHAPTER 6

EDITABLE DOCUMENTS

The EPROOT, EPFLAT and EPSEG classes provide the means of storing, reading and editing, in memory, the text of a document. The text must always have a terminating zero but may otherwise be of any length up to a maximum of 65535 characters, subject to memory constraints.

The methods support the use of an instance of (a subclass of) EPFLAT or EPSEG as a clipboard. Text may be cut or copied into such a clipboard from a document or pasted from a clipboard into a document.

Document content

The concepts of words, paragraphs and blocks are built into the document content model, where:

- a paragraph is any sequence of characters delimited by a paragraph delimiter. A paragraph delimiter is an ASCII 0, 1, 2 or 3, a value of 0 being by far the most commonly used. The final paragraph delimiter in the document is always an ASCII zero.
- a word is any sequence of characters delimited by one or more word delimiter characters. A word delimiter is either a paragraph delimiter, or a whitespace character (for which `p_isspace` returns TRUE).
- a block is a sequence of paragraphs delimited by paragraphs that are either empty or, if not empty, contain only whitespace characters.

The terminating ASCII zero is generally not regarded as part of the editable content. It is not, for example, included in the document length as returned by an `ep_sense_len` method.

Its presence is fundamental to the operation of all classes that subclass EPROOT and it should never be deleted.

Addressable character positions

A position in an EPFLAT or EPSEG document is considered, in general, to mark the point between two adjacent characters. Thus, character position 3 is interpreted as being between the third and fourth characters. Character position zero is before the first character in the document.

Inserting at position 9, say, will insert text after the ninth character: deleting the characters between positions 3 and 5 will delete the fourth and fifth characters.

The last addressable character position is immediately before the final paragraph delimiter (always an ASCII zero).

Usage

Instances of (subclasses of) EPFLAT and EPSEG are widely used in all SIBO machines to contain editable text. Examples range from the text in a dialog edit box, to the text of a word processor document.

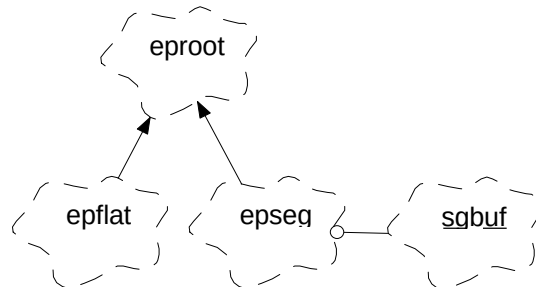
In general, The EPFLAT class should be used for small amounts of text, or for documents with a limited variability of content, whereas the EPSEG class should be used for larger documents, or for those with a wide dynamic content range. (Compare this with the recommended use of the VAFLAT and VASEG variable array classes.)

Precursors

An understanding of the following topics would prove helpful:

- the `p_enter` and `p_leave` error handling services
- for EPSEG, the SGBUF segmented buffer class

Class diagram



EPROOT

EPROOT	
maxlen	
<code>ep_set_text</code>	<code>ep_sense_text</code>
<code>ep_scan_word</code>	<code>ep_capacity</code>
<code>ep_word_count</code>	
<code>ep_scan_para</code>	<i>ep_compress</i>
<code>ep_para_count</code>	<i>ep_init</i>
<code>ep_scan_block</code>	<i>ep_sense_len</i>
<code>ep_add_para</code>	<i>ep_sense_chars</i>
<code>ep_copy_indent</code>	<i>ep_back_chars</i>
<code>ep_copy_to_front</code>	<i>ep_insert</i>
<code>ep_copy_to_back</code>	<i>ep_delete</i>
<code>ep_paste</code>	<i>ep_extract</i>
<code>ep_mod_chars</code>	<i>ep_clear</i>

The EPROOT abstract class provides the basic methods for manipulating the text of a document.

Class definition

Defined in sub-category file *edit.cl* (generated header file *edit.g*).

```

CLASS eproot root
Editable paragraphs (abstract class)
{
  ADD ep_set_text          Set the contents, replacing any previous contents
  ADD ep_scan_word         Scan by words
  ADD ep_word_count        Return word count
  ADD ep_scan_para         Scan by paragraphs
  ADD ep_para_count        Return paragraph count
  ADD ep_scan_block        Scan by blocks
  ADD ep_add_para          Append a paragraph given its content
  ADD ep_copy_indent        Copy indent from previous paragraph
  ADD ep_copy_to_front     Copy range to front of provided clipboard
  ADD ep_copy_to_back      Copy range to back of provided clipboard
  ADD ep_paste             Paste from provided clipboard, update position
  ADD ep_mod_chars         Various mods to a range of characters
  ADD ep_sense_text        Copy out all the data and return its length
  ADD ep_capacity=p_dummy  Adjust storage to stated capacity
  DEFER ep_compress        Compress storage to minimum possible
  DEFER ep_init            Prepare an empty editable object
  DEFER ep_sense_len       Return length of data
  DEFER ep_sense_chars     Provide pointer to following contiguous data
  DEFER ep_back_chars      Provide pointer to previous contiguous data
  DEFER ep_insert          Insert block of characters
  DEFER ep_delete          Delete range of text
  DEFER ep_extract         Extract data into buffer
  DEFER ep_clear           Empty the object of all contents
  CONSTANTS
  {
    EP_SCAN_BACKWARDS 0x01 Scan backwards if set
    EP_SCAN_STAY      0x02 Will stay put if already at a boundary
    EP_SCAN_TO_BEGIN  0x04 Stops at beginning of unit
    EP_SCAN_TO_END     0x08 Stops at end of unit
    EP_SCAN_JOIN_DELIM 0x10 Sequence of delimiters count as one
    EP_SCAN_NOT_TO_END 0x20 Do not scan past the final terminator
    EP_RETURN_LENGTH -1 special meaning for ep_sense_chars
    EP_MOD_TOLOWER 0    fold to lower case
    EP_MOD_TOUPPER 1    fold to upper case
    EP_MOD_WRAP 2      join paragraphs by converting zeros to spaces
  }
  PROPERTY
  {
    UWORD maxlen;          Maximum permissible length of text
  }
}

```

Property

`eproot.maxlen` the maximum permissible length of the text. All subclasses must set this field, normally in an `ep_init` method

EPROOT methods

EP_SET_TEXT

Set content

```
VOID ep_set_text(TEXT *buf, UINT len);
```

Replace any existing text with a single paragraph containing `len` characters copied from `*buf`. The value of `len` should exclude any trailing paragraph terminator.

Calls `p_leave` for out of memory errors. In this event, *none* of the existing text will have been replaced.

★ EP_SCAN_WORD**Scan by word**

```
UINT ep_scan_word(UWORD *ppos, UINT flags);
```

Scan, from character position **ppos*, by one word, updating **ppos* to the new character position and returning the number of characters skipped by the scan. There is no need for the initial position to be at a word boundary.

The value of *flags* may be any combination of the following:

EP_SCAN_BACKWARDS	Scan backwards if set, otherwise scan forwards
EP_SCAN_STAY	Do not scan if already at a word boundary
EP_SCAN_TO_BEGIN	Scan to the beginning of a word
EP_SCAN_TO_END	Scan to the end of a word
EP_SCAN_JOIN_DELIM	Treat a sequence of word delimiters as a single delimiter
EP_SCAN_NOT_TO_END	Do not scan past the final terminator

★ EP_WORD_COUNT**Count words**

```
UINT ep_word_count(VOID);
```

Return a count of the total number of words in the document.

★ EP_SCAN_PARA**Scan by paragraph**

```
UINT ep_scan_para(UWORD *ppos, UINT flags);
```

Scan, from character position **ppos*, by one paragraph, updating **ppos* to the new character position and returning the number of characters skipped by the scan. There is no need for the initial position to be at a paragraph boundary.

The value of *flags* may be any combination of the following:

EP_SCAN_BACKWARDS	Scan backwards if set, otherwise scan forwards
EP_SCAN_STAY	Do not scan if already at a paragraph boundary
EP_SCAN_TO_BEGIN	Scan to the beginning of a paragraph
EP_SCAN_TO_END	Scan to the end of a paragraph
EP_SCAN_JOIN_DELIM	Treat a sequence of paragraph delimiters as a single delimiter
EP_SCAN_NOT_TO_END	Do not scan past the final terminator

★ EP_PARA_COUNT**Count paragraphs**

```
UINT ep_para_count(VOID);
```

Return a count of the total number of paragraphs in the document.

★ EP_SCAN_BLOCK**Scan by block**

```
UINT ep_scan_block(UWORD *ppos, UINT flags);
```

Scan, from character position **ppos*, by one block, updating **ppos* to the new character position and returning the number of characters skipped by the scan.

The initial position is assumed to be at a paragraph boundary.

The value of `flags` may be any combination of the following:

<code>EP_SCAN_BACKWARDS</code>	Scan backwards if set, otherwise scan forwards
<code>EP_SCAN_STAY</code>	Do not scan if already at a block boundary
<code>EP_SCAN_TO_BEGIN</code>	Scan to the beginning of a block
<code>EP_SCAN_TO_END</code>	Scan to the end of a block
<code>EP_SCAN_JOIN_DELIM</code>	Treat a sequence of block delimiters as a single delimiter
<code>EP_SCAN_NOT_TO_END</code>	Do not scan past the final terminator

EP_ADD_PARA

Append a paragraph

```
VOID ep_add_para(TEXT *buf, UINT len);
```

Append a paragraph containing `len` bytes of text copied from `*buf`.

The text in `*buf` is appended more efficiently if it is supplied as a zero terminated string - when `len` should be equal to `p_strlen(buf)` - but the terminating zero is not mandatory.

Calls `p_leave`, without inserting anything, if there is insufficient memory available to append the paragraph.

EP_COPY_INDENT

Copy whitespace indentation

```
UINT ep_copy_indent(UWORD *ppos);
```

Copy any leading whitespace from the previous paragraph (which is assumed to exist) to the character position indicated by `*ppos`. This position will normally be the start of a paragraph.

The value of `*ppos` is incremented by the number of characters inserted.

Returns the number of characters inserted.

Inserts nothing and calls `p_leave` if there is insufficient memory to perform the insertion.

Returns zero if the insertion is successful. The method is suitable for being called under the protection of `p_enter`.

EP_COPY_TO_FRONT

Copy range to front of clipboard

```
INT ep_copy_to_front(UINT pos1, UINT pos2, PR_ROOT *clip);
```

Copy the range of characters between positions `pos1` and `pos2` to the front (position zero) of the clipboard `clip`, where `clip` is assumed to be the handle of an instance of (a subclass of) `EPROOT` or `EPSEG`.

Inserts nothing in the clipboard and calls `p_leave` if there is insufficient memory to perform the insertion.

Returns zero if the insertion is successful. The method is suitable for being called under the protection of `p_enter`.

EP_COPY_TO_BACK

Copy range to back of clipboard

```
INT ep_copy_to_back(UINT pos1, UINT pos2, PR_ROOT *clip);
```

Copy the range of characters between positions `pos1` and `pos2` to the back (before the terminating zero) of the clipboard `clip`, where `clip` is assumed to be the handle of an instance of (a subclass of) `EPROOT` or `EPSEG`.

Inserts nothing in the clipboard and calls `p_leave` if there is insufficient memory to perform the insertion.

Returns zero if the insertion is successful. The method is suitable for being called under the protection of `p_enter`.

EP_PASTE**Insert from clipboard**

```
INT ep_paste(UWORD *ppos, PR_EPROOT *clip);
```

Insert the contents of the clipboard `clip`, assumed to be the handle of an instance of (a subclass of) `EPROOT` or `EPSEG`, at position `*ppos`. The value of `*ppos` is updated to the end of the inserted text.

Inserts nothing and calls `p_leave` if there is insufficient memory to perform the insertion.

Returns zero if the insertion is successful. The method is suitable for being called under the protection of `p_enter`.

★ EP_MOD_CHARS**Modify characters in a range**

```
VOID ep_mod_chars(UINT pos1, UINT pos2, UINT mod);
```

Modify the characters between positions `pos1` and `pos2`. The modification depends on the value of `mod`, which may be one of:

<code>EP_MOD_Toupper</code>	convert characters to upper case, with <code>p_toupper</code>
<code>EP_MOD_Tolower</code>	convert characters to lower case, with <code>p_tolower</code>
<code>EP_MOD_WRAP</code>	convert each paragraph delimiter character to a space (ASCII 32) character

★ EP_SENSE_TEXT**Copy text to buffer**

```
UINT ep_sense_text(TEXT *buf);
```

Copy the entire text of the document, including the terminating zero, to `*buf`. It is the user's responsibility to ensure that the buffer is of sufficient length.

Returns the number of characters copied, excluding the terminating zero.

★ EP_CAPACITY**Set document capacity**

```
VOID ep_capacity(UINT len);
```

This method does nothing, which is the appropriate action for subclasses using segmented storage (that is, `EPSEG` or a subclass of `EPSEG`).

Subclasses using storage in a single allocated segment should subclass this method (see, for example, `EPFLAT`'s `ep_capacity` method).

Deferred EPROOT methods

The deferred methods, listed below, are all fully described in the following documentation of the `EPFLAT` and `EPSEG` classes.

<code>EP_INIT</code>	Initialise
<code>EP_SENSE_LEN</code>	Sense document length
<code>EP_SENSE_CHARS</code>	Sense characters forwards
<code>EP_BACK_CHARS</code>	Sense characters backwards
<code>EP_INSERT</code>	Insert characters
<code>EP_EXTRACT</code>	Copy out characters
<code>EP_DELETE</code>	Delete characters
<code>EP_CLEAR</code>	Clear the document
<code>EP_COMPRESS</code>	Compress allocated storage

EPFLAT

EPROOT		EPFLAT
maxlen		alen gran len buf
ep_set_text	ep_sense_text	destroy
ep_scan_word	~~ep_capacity~~	ef_granularity
ep_word_count		ef_sense_buf
ep_scan_para	~~ep_compress~~	ep_init
ep_para_count	~~ep_init~~	ep_sense_len
ep_scan_block	~~ep_sense_len~~	ep_sense_chars
ep_add_para	~~ep_sense_chars~~	ep_back_chars
ep_copy_indent	~~ep_back_chars~~	ep_insert
ep_copy_to_front	~~ep_insert~~	ep_extract
ep_copy_to_back	~~ep_delete~~	ep_delete
ep_paste	~~ep_extract~~	ep_clear
ep_mod_chars	~~ep_clear~~	ep_compress
		ep_capacity

EPFLAT stores the document text contiguously in a single allocated cell.

This class is intended for use either to store relatively small amounts of text, or where the dynamic range of the document size is small.

A typical example is to store the text of a dialog edit box.

Class definition

Defined in sub-category file *edit.cl* (generated header file *edit.g*).

```

CLASS epflat eproot
  Editable paragraphs - stored in a single allocated cell
  {
    REPLACE destroy
    REPLACE ep_init
    REPLACE ep_sense_len
    REPLACE ep_sense_chars
    REPLACE ep_back_chars
    REPLACE ep_insert
    REPLACE ep_extract
    REPLACE ep_delete
    REPLACE ep_clear
    REPLACE ep_compress
    REPLACE ep_capacity
    ADD ef_granularity          Overwrite the default granularity
    ADD ef_sense_buf           More convenient than ep_sense_chars

    PROPERTY
    {
      UWORD alen;      length of allocated cell
      UWORD gran;      granularity to grow by
      UWORD len;        length of text stored
      TEXT *buf;        address of text buffer
    }
  }

```

Property

epflat.alen	the size, in bytes, of the allocated cell. This should not be accessed by a subclass.
epflat.gran	the granularity, in bytes. The allocated cell is grown, when necessary, in multiples of this value. This should not be accessed by a subclass.
epflat.len	the number of bytes of stored text. This should be treated as a read-only field by a subclass.
epflat.buf	a pointer to the start of the buffer containing the text. This should be treated as a read-only field by a subclass.

EPFLAT methods

★ DESTROY

Destroy

```
VOID destroy(VOID);
```

Free the allocated buffer and supersend the DESTROY message.

EP_INIT

Initialise

```
VOID ep_init(UINT maxlen);
```

Initialise the instance of EPFLAT to be suitable to contain up to `maxlen` bytes of text (typically the text will not exceed that which can be displayed on a single line).

Sets `eproot.maxlen` to `maxlen` and `epflat.gran` to a default value of eight bytes. Allocates a minimum size buffer and inserts a single NULL character.

Calls `p_leave` if there is insufficient memory to allocate the buffer.

★ EP_SENSE_LEN

Sense document length

```
INT ep_sense_len(VOID);
```

Returns the number of characters in the document, excluding the terminating zero.

★ EP_SENSE_CHARS

Sense characters forwards

```
UINT ep_sense_chars(TEXT **pbuf, UINT pos, UINT n);
```

Write, to `*pbuf`, the address of the character at position `pos`.

Returns the smaller of `n` and the number of contiguously stored characters at `*pbuf`, *including* the zero that terminates the document.

Note that for compatibility with the EPSEG class, the interface to this method does not assume that the entire text of the document is stored contiguously.

★ EP_BACK_CHARS

Sense characters backwards

```
UINT ep_back_chars(TEXT **pbuf, UINT pos, UINT n);
```

Write, to `*pbuf`, the address of the character whose position is `n` characters in front of position `pos`; if `n` is greater than the number of characters in front of position `pos`, the address of the first character is taken.

Returns the number of contiguous characters at `*pbuf` up to, but not including, the character at position `pos`. This value will not exceed `n`.

Note that, for compatibility with the EPSEG class, the interface to this method does not assume that the entire text of the document is stored contiguously.

EP_INSERT

Insert characters

```
INT ep_insert(UINT pos, TEXT *buf, UINT len);
```

Insert `len` characters from `*buf`, at character position `pos`.

If `pos` is `-1`, the characters are inserted at the end of the document (before the terminating zero). This feature is not replicated in the `ep_insert` method of the EPSEG class.

Inserts nothing and calls `p_leave` if there is insufficient memory to perform the insertion (with an `E_GEN_NOMEMORY` error) or if the resulting document length were to exceed `eproot.maxlen` (with an `E_GEN_OVER` error).

Returns zero if the insertion is successful. The method is suitable for being called under the protection of `p_enter`.

★ EP_EXTRACT**Copy out characters**

```
VOID ep_extract(UINT pos, TEXT *buf, UINT len);
```

Copy len characters to *buf, starting with the character at position pos.

The user is responsible for ensuring that the buffer is of sufficient length to contain the text.

Note that no check is made to ensure that len characters are available at position pos.

★ EP_DELETE**Delete characters**

```
VOID ep_delete(UINT pos1, UINT pos2);
```

Delete the characters between positions pos1 and pos2, without making any attempt to reduce the size of the allocated buffer.

If pos2 is -1 then all characters between position pos1 and the end of the document are cleared.

If pos2 is less than or equal to pos1 then the method does nothing.

★ EP_CLEAR**Clear the document**

```
VOID ep_clear(VOID);
```

Remove any text. This leaves a document containing a single terminating zero.

No attempt is made to reduce the size of the allocated buffer.

EP_COMPRESS**Compress allocated storage**

```
VOID ep_compress(VOID);
```

Reduce the size of the allocated cell to the minimum required to contain the document, subject to the restraint of the current value of `epflat.gran`. The cell will never be smaller than `epflat.gran` bytes in length.

This method can only fail (by calling `p_leave`) if it follows an `EF_GRANULARITY` message that alters the granularity such that the `ep_compress` method causes the allocated cell to increase in size. In general (and certainly if the `ef_granularity` method is never called) it is safe to assume that this method will never fail.

EP_CAPACITY**Set document capacity**

```
VOID ep_capacity(UINT len);
```

Set the document capacity (using `f_realloc`) to len bytes, rounded up to be a multiple of `epflat.gran`.

Calls `p_leave` if there is insufficient memory to reallocate the cell.

★ EF_GRANULARITY**Set buffer granularity**

```
VOID ef_granularity(UINT gran);
```

Set `epflat.gran` to gran or, if gran is zero, set `epflat.gran` to 1.

No attempt is made to alter the current size of the allocated buffer which may, therefore, be incompatible with the new granularity.

★ EF_SENSE_BUF**Sense start of data**

```
UINT ef_sense_buf(TEXT **pbuf);
```

Write, to *pbuf, the address of the first byte of the document text, returning the length of the text, excluding the terminating zero.

This method, unlike `ep_sense_chars`, takes advantage of the fact that the whole of the text is stored contiguously.

EPSEG

EPROOT		EPSEG
maxlen		b
ep_set_text	ep_sense_text	ep_init
ep_scan_word	ep_capacity	ep_sense_len
ep_word_count		ep_sense_chars
ep_scan_para	~~ep_compress~~	ep_back_chars
ep_para_count	~~ep_init~~	ep_insert
ep_scan_block	~~ep_sense_len~~	ep_extract
ep_add_para	~~ep_sense_chars~~	ep_delete
ep_copy_indent	~~ep_back_chars~~	ep_clear
ep_copy_to_front	~~ep_insert~~	ep_compress
ep_copy_to_back	~~ep_delete~~	
ep_paste	~~ep_extract~~	
ep_mod_chars	~~ep_clear~~	

EPSEG stores the document text in a segmented buffer using an instance of SGBUF.

This class is intended for use either to store large amounts of text, or where the dynamic range of the document size is potentially large.

A typical example is to store the text of a text processor document.

Class definition

Defined in sub-category file *edit.cl* (generated header file *edit.g*).

```

CLASS  epseg  eproot
Editable paragraphs - segmented storage
{
  REPLACE ep_init
  REPLACE ep_sense_len
  REPLACE ep_sense_chars
  REPLACE ep_back_chars
  REPLACE ep_insert
  REPLACE ep_extract
  REPLACE ep_delete
  REPLACE ep_clear
  REPLACE ep_compress
  PROPERTY 1
    {
      PR_SGBUF *b;    handle of buffers data
    }
}

```

Property

epseg.b	the handle of an instance of the SGBUF segmented buffer class in which the text is stored. It should be regarded as read only.
---------	--

EPSEG methods

EP_INIT

Initialise

```
VOID ep_init(UINT maxlen);
```

Initialise, by setting eproot.maxlen to maxlen, creating an instance of SGBUF and initialising it with a fixed granularity of 64 bytes and then inserting a single NULL character.

Calls p_leave if there is insufficient memory.

★ EP_SENSE_LEN Sense document length

```
UINT ep_sense_len(VOID);
```

Returns the number of characters in the document, excluding the terminating zero.

★ EP_SENSE_CHARS Sense characters forwards

```
UINT ep_sense_chars(TEXT **pbuf, UINT pos, UINT n);
```

Write, to *pbuf, the address of the character at position pos.

Returns the smaller of n and the number of contiguously stored characters at *pbuf, *including* the zero that terminates the document.

★ EP_BACK_CHARS Sense characters backwards

```
UINT ep_back_chars(TEXT **pbuf, UINT pos, UINT n);
```

Write, to *pbuf, the address of the character whose position is n characters in front of position pos; if n is greater than the number of characters in front of position pos, the address of the first character is taken.

Returns the number of contiguous characters at *pbuf up to, but not including, the character at position pos. This value will not exceed n.

EP_INSERT Insert characters

```
INT ep_insert(UINT pos, TEXT *buf, UINT len);
```

Insert len characters from *buf, at character position pos.

Inserts nothing and calls p_leave if there is insufficient memory to perform the insertion (with an E_GEN_NOMEMORY error) or if the resulting document length were to exceed eproot.maxlen (with an E_GEN_OVER error).

Returns zero if the insertion is successful. The method is suitable for being called under the protection of p_enter.

★ EP_EXTRACT Copy out characters

```
VOID ep_extract(UINT pos, TEXT *buf, UINT len);
```

Copy len characters to *buf, starting with the character at position pos. The position pos must lie within the range of the segmented buffer or else p_panic(P_PANIC_SGBUF_1) will be called.

The user is responsible for supplying a buffer of sufficient length to contain the text.

★ EP_DELETE Delete characters

```
VOID ep_delete(UINT pos1, UINT pos2);
```

Delete the characters between positions pos1 and pos2, by sending the owned instance of SGBUF an SB_DELETE message.

If the position pos1 lies outside the range of the segmented buffer, p_panic(P_PANIC_SGBUF_1) will be called. Similarly if the position pos2 lies outside the range of the segmented buffer, p_panic(P_PANIC_SGBUF_2) will be called.

★ EP_CLEAR Clear the document

```
VOID ep_clear(VOID);
```

Remove any text (by sending the owned instance of SGBUF an SB_DELETE message) leaving a document containing a single terminating zero.

★ EP_COMPRESS Compress allocated storage

```
VOID ep_compress(VOID);
```

Reduce the size of the owned instance of SGBUF to the minimum required to contain the document, consistent with its granularity of 64 bytes, by sending it an SB_COMPRESS message.

CHAPTER 7

RESOURCE FILES

RSCFILE
pcb ix offset hftree
destroy rs_init rs_read rs_read_buf

The RSCFILE class provides a set of services to access the contents of a resource file. Resource files are described in the *Resource Files* chapter of the *Additional System Information* manual. As mentioned there, a resource file may be embedded in an image file and may optionally be Huffman code compressed.

An application that needs a resource file and also uses an application manager (APPMAN) will generally pass the `FLG_APPMAN_RSCFILE` flag to the APPMAN `am_init` method. This causes an instance of the RSCFILE class to be created automatically. The application can then read resources by means of the application manager's `am_load_resource` and `am_load_res_buf` methods which offer greater functionality than the RSCFILE `rs_read` and `rs_read_buf` methods. Such users will not require detailed knowledge of the RSCFILE class.

Precursors

The reader is assumed to understand:

- the `p_enter` and `p_leave` error handling services

Class definition

The RSCFILE class subclasses ROOT and is defined in the sub-category file *appman.cl* (with generated header file *appman.g*).

```
CLASS rscfile root
Basic access to a resource file which may be embedded in an image file
{
  REPLACE destroy          Close channel and destroy
  ADD rs_init              Open a resource file
  ADD rs_read              Read a record into an allocated cell
  ADD rs_read_buf          Read a record into a buffer
  TYPES
  {
    typedef struct
    {
      UWORD pos;          Index file position
      UWORD len;          Index length
    } PR_RSCFILE_HEAD;
  }
  PROPERTY
  {
    UBYTE *pcb;          resource file channel
    PR_RSCFILE_HEAD ix;  header containing index position and length
    UWORD offset;        file offset of start of resource data
    ULONG hftree;        Huffman tree data
  }
}
```

Property

<code>rscfile.pcb</code>	the currently opened resource file channel handle. It should not be accessed by a subclass.
<code>rscfile.ix</code>	the resource file header, containing the index position and length. It should not be accessed by a subclass.
<code>rscfile.offset</code>	the offset from the start of the file to the start of the resource data (zero unless the file is embedded in an image file). It should not be accessed by a subclass.
<code>rscfile.hftree</code>	the Huffman tree data. It should not be accessed by a subclass.

RSCFILE methods**J DESTROY****Destroy**

```
VOID destroy(VOID);
```

Close the currently opened resource file and supersend the DESTROY message.

RS_INIT**Initialise**

```
INT rs_init(UBYTE *pname);
```

Open a channel to a resource file. The string pointed to by `*pname` should be the name of either the resource file itself, or of an image file containing, in its second add-file slot, an embedded resource file. (See the *Resource Files* chapter of the *Additional System Information* manual.)

Writes the appropriate values to `rscfile.offset`, `rscfile.ix` and, if the resources are Huffman encoded, `rscfile.hftree`.

Calls `p_leave` on error, otherwise returns zero. The method is suitable for being called under the protection of `p_enter`.

RS_READ**Allocate buffer and read resource**

```
INT rs_read(INT rid, UBYTE **ppdata);
```

Allocate a buffer and read into it the resource with resource id `rid`.

The length of the resource is read from the file and a buffer of this length is allocated. The resource is then read into this buffer and the address of the buffer is written to `*ppdata`. If the resource file record is Huffman encoded then it is decoded before being written to the buffer.

The method calls `p_leave` on any error (which will be either a memory allocation error or an error while attempting to read the file). It is guaranteed that, on error, no memory will have been allocated and nothing will have been written to `*ppdata`.

Returns the length of the resource, including the terminating zero if the resource is a string.

RS_READ_BUF**Read resource**

```
INT rs_read_buf(INT rid, UBYTE *buf);
```

Read the resource with resource id `rid` into the buffer pointed to by `buf`. If the resource file record is Huffman encoded then it is decoded before being written to the buffer. It is the user's responsibility to ensure that the buffer is of sufficient length to contain the resource.

The method calls `p_leave` on error (which will be an error while attempting to read the file).

Returns the length of the resource, including the terminating zero if the resource is a string.

CHAPTER 8

BINARY FILE MANAGEMENT

The classes described in this chapter provide methods for reading and writing signed binary files.

Such files contain a 22 byte standard header consisting of:

- a 16 byte file signature (all 16 bytes are significant)
- a 2 byte file version number
- a 2 byte offset from the start of the file to the end of the header (to allow for future expansion of the header)
- a 2 byte runtime version number

Each version number is a hexadecimal number in the form XYF , where:

X is the major version number (4 bits)
YY is the minor version number (8 bits)
F is the release type, A (alpha) B (beta) or F (final)

For example, 0x123A is an alpha release of version 1.23.

The runtime version number is intended to specify the minimum version of runtime software (for example, OPL) that is required to process the file. If this field is not used it should be set to zero.

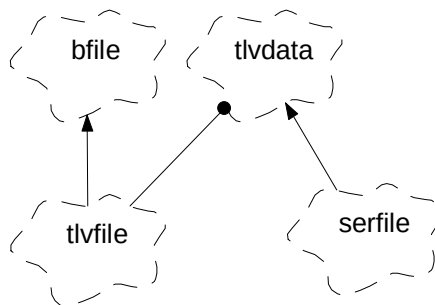
Database files are a particular type of signed binary file. Further information relating to this type of file may be found in the *Database Files* chapter of the *PLIB Reference* manual and in the *ISAM Reference* manual.

Precursors

An understanding of the following topics would prove helpful:

- the PLIB binary file services, as described in the *Files* chapter of the *PLIB Reference* manual.
- the `p_enter` and `p_leave` error handling services

Class diagram



BFILE

BFILE
pcb rbuf rlen offset
destroy fi_close fi_read fl_set_buf_len fl_sense_data fi_open fl_rewind

The methods of the BFILE class provide the basic means of creating, opening, validating and reading signed binary files with arbitrary content.

BFILE must be subclassed to provide additional methods if it is necessary to write to the file.

Class definition

Defined in sub-category file *tlvfile.cl* (generated header file *tlvfile.g*).

```
CLASS bfile root
{
  REPLACE destroy      Close file then destroy
  ADD fi_close         Free any buffers
  ADD fi_read          Read from file into internal buffer
  ADD fl_set_buf_len    Ensure internal buffer is at least len bytes
  ADD fl_sense_data     Get length and address of data
  ADD fi_open           Open binary file and check signature
  ADD fl_rewind         Reposition to first byte after header
  CONSTANTS
  {
    OP_BFILE_ID_SIZE  16    size of text ID
  }
  TYPES
  {
    typedef struct
    {
      TEXT fid[OP_BFILE_ID_SIZE];  plain text application ID
      UWORD vers;                  file version number
      UWORD offset;                abs. file offset to end of header
      UWORD rtvers;               minimum runtime version
    } OP_BFILE_FSIG;
  }
  PROPERTY
  {
    UBYTE *pcb;      File channel
    UBYTE *rbuf;     Allocated record buffer
    UWORD rlen;      Length of data in read buffer
    UWORD offset;
  }
}
```

Property

bfile.pcb	the handle of a currently open file, or NULL. It should not be accessed by any subclass.
bfile.rbuf	a pointer to an allocated buffer into which file data is read.
bfile.rlen	the number of bytes of valid data in the allocated buffer.
bfile.offset	the byte offset from the start of the current file to the first byte after the file signature.

BFILE methods

★ DESTROY

Destroy

```
VOID destroy(VOID);
```

Send an FI_CLOSE message and then supersend the DESTROY message.

FI_OPEN

Open file

```
INT fi_open(TEXT *pname, UINT mode, OP_BFILE_SIG *psig);
```

Open the binary file whose full file specification is pointed to by `pname`. The value of `mode` may be any combination of the PLIB file mode flags appropriate for a binary file (it must include `P_FSTREAM` or, more rarely, `P_FSTREAM_TEXT`).

If a new file is being opened (`mode` contains either `P_FCREATE` or `P_FREPLACE`) the file signature at `*psig` is written to the newly opened file. Otherwise, the file signature is read from the file and compared with the file signature at `*psig`. If the file signature read from the file is the wrong length, or if the two 16-byte file IDs do not match exactly (all sixteen bytes are compared) `p_leave(E_FILE_INVALID)` is called. The validation of the remaining signature fields will vary with the application and is left to the caller. To facilitate this validation, the signature read from the file is written to `*psig`.

Once the file has been successfully opened, the file offset to the first byte following the file header (`psig->offset`) is written to `bfile.offset`.

Returns zero if successful, or a negative error number for any error other than the `E_FILE_INVALID` error described above.

★ FI_CLOSE

Close file

```
VOID fi_close(VOID);
```

Close any open file, freeing any allocated buffer.

It is safe to send an FI_CLOSE message even if there is no open file.

FI_READ

Read from file

```
INT fi_read(UINT len);
```

Read `len` bytes from the current position in the currently open file into the allocated buffer.

Sends an FL_SET_BUF_LEN message to ensure that the buffer has room for at least `len` bytes before reading the data. If this fails, `p_leave(E_GEN_NOMEMORY)` is called.

If the read is successful, sets `bfile.rlen` to contain the number of bytes read into the buffer.

Returns the number of bytes read, or a negative error.

FL_SET_BUF_LEN

Set buffer size

```
VOID fl_set_buf_len(UINT len);
```

Reallocate, if necessary, the allocated buffer pointed to by `bfile.rbuf` to ensure that it has room for at least `len` bytes. This method will never reduce the size of the buffer.

Calls `p_leave` if there is insufficient memory to reallocate the buffer.

★ FL_SENSE_DATA**Sense record data**

```
UINT fl_sense_data(UBYTE **pbuf);
```

Write to **pbuf* the address of the allocated buffer and return the length of the data last read into it by the *fi_read* method.

The return value will be zero and the value written to **pbuf* will be `NULL` if there is no currently open file, or if no `FI_READ` message has been received.

FL_REWIND**Reposition to start**

```
VOID fl_rewind(VOID);
```

Set the current file position to the first byte after the file header.

TLVFILE

BFILE	TLVFILE
pcb rbuf rlen offset	typmask hdlen hdtype pos fpos
destroy fi_close fi_read fl_set_buf_len fl_sense_data ~~fi_open~~ ~~fl_rewind~~	fi_open fl_rewind fl_write_rec fl_delrec fl_count fl_read_by_type fl_set_rec fl_sense_rec fl_replace

The `TLVFILE` class provides support for signed binary files that contain type-length-value (TLV) records.

Following the standard header, the file is considered to be made up of records, each of which has a 2 byte header specifying the record type and length. The most significant nibble of the word contains the record type, in the range 0-15. The remaining three nibbles contain the record length and is, therefore, restricted to a maximum of 4K bytes.

Records of type 0 are considered to be deleted records. Records of type 15 (0x0f) are reserved to represent non-valid records and are treated as though they are deleted records.

For efficient record access, the record number of the next record to be read and the current file position are stored in property.

TLV files are designed to be *Flash-friendly*, i.e. they may be stored and manipulated efficiently in Flash SSDs (or any other EPROM medium). A TLV file stored on such a medium may be modified by appending, deleting or replacing records without having to make a new copy of the entire file.

Although TLV files may contain in excess of 4,000,000,000 records, `TLVFILE` is ideally suited to manipulating files which contain a relatively small number of records. If the file contains a large number of records, operations which involve non-sequential access may take an extended time to return.

Database files are a form of TLV file with a particular file signature header and specific record content. Alternative means of manipulating such files are described in the *Database Files* chapter of the *PLIB Reference* manual and also in the *ISAM Reference* manual.

Class definition

Defined in sub-category file *tlvfile.cl* (generated header file *tlvfile.g*).

```

CLASS tlvfile bfile
{
  REPLACE fi_open           Open file
  REPLACE fl_rewind         Reposition to first record, reset property
  ADD fl_write_rec          Write a TLV record
  ADD fl_delrec             Delete a record
  ADD fl_count              Get record count
  ADD fl_read_by_type       Read record of specified type
  ADD fl_set_rec            Set the current record number
  ADD fl_sense_rec          Sense the current record number
  ADD fl_replace            Replace a record

  CONSTANTS
  {
    TLV_TYPE_UNKNOWN      0x10
    TLV_TYPE_INVALID      0x0f
    TLV_TYPE_DELETED      0x00
    TLV_TYPE_NORMAL       0x01
    TLV_TYPE_FIELDS       0x02
    TLV_TYPE_SHIFT        12
    TLV_TYPE_MASK         0xf000
  }

  TYPES
  {
    typedef struct
    {
      OP_BFILE_FSIG fsig;   Binary file signature
      UWORD types;         Valid types
    } OP_TLVFILE;
    typedef struct
    {
      UBYTE *buf;
      UINT len;
      INT type;
    } OP_TLV_REC;
  }

  PROPERTY
  {
    UWORD typmask; valid record type mask
    UWORD hdlen;   length of header
    UWORD hdtype;  type from header
    ULONG pos;     Next record to be read
    ULONG fpos;    Current file position, for validation
  }
}

```

Property

<code>tlvfile.typmask</code>	which record types are considered valid. For example, the value 0x32 (bits 1, 4 and 5 set) indicates that records of type 1, 4 and 5 are valid. Records of other types are treated as if they do not exist.
<code>tlvfile.hdlen</code>	the current record length as decoded from the first word of the record. It should not be accessed by a subclass.
<code>tlvfile.hdtype</code>	the current record type as decoded from the first word of the record. It should not be accessed by a subclass.
<code>tlvfile.pos</code>	the current record number that corresponds to the file position, <code>tlvfile.fpos</code> . It should not be accessed by a subclass.
<code>tlvfile.fpos</code>	the current file position. It should not be accessed by a subclass.

TLVFILE methods

All methods which read a record assume that the current file position is at the start of a record.

FI_OPEN

Open TLV file

```
INT fi_open(TEXT *fspec, UINT mode, OP_TLVFILE *psig):
```

Open the TLV file whose full file specification is pointed to by `fspec`. The value of `mode` may be any combination of the PLIB file mode flags appropriate for a binary file (it must include `P_FSTREAM`).

Opens the file by supersending the `FI_OPEN` message which, if opening an existing file, validates the 16-byte file signature ID and overwrites `psig->fsig` (but not `psig->types`) with the signature read from the file.

Calls `p_leave` if the file signature validation fails.

Sets `tlvfile.typmask` to the value of `psig->types` and sends itself an `FL_REWIND` message to position to the start of the first record.

Returns zero if successful or error values as returned by the `BFILE` superclass.

FL_REWIND

Reposition to start

```
VOID fl_rewind(VOID);
```

Position to the first byte following the file signature.

Supersends the `FL_REWIND` message and then sets `tlvfile.fpos` to `bfile.offset`, and `tlvfile.pos` to zero.

Calls `p_leave` on error.

FL_COUNT

Count records

```
VOID fl_count(ULONG *pcount);
```

Write to `*pcount` the number of valid records (those whose types are specified by `tlvfile.typmask`) in the file.

Counts the records by scanning the entire file and then sends an `FL_REWIND` message to reposition to the first record.

Calls `p_leave` on error.

FL_WRITE_REC

Write a record

```
VOID fl_write_rec(UBYTE *buf, UINT len, UINT type);
```

Append a new record of type `type`, containing the first `len` bytes of the data pointed to by `buf`.

If any error occurs, the record is either not written or is marked as not being a valid record.

All errors result in `p_leave` being called.

FL_SET_REC

Set current record

```
INT fl_set_rec(UINT lsw, UINT msw);  
INT fl_set_rec(ULONG recnum); (conceptual)
```

Position to, and read into the internal buffer, record number `recnum` (counting only records of types specified by `tlvfile.typmask`). The `ULONG recnum` is actually passed in the message as two `UINT` parameters, `lsw` (least significant word) and `msw` (most significant word).

Returns the positive record type if successful, or `E_FILE_EOF` if reading past the end of the file. Other file errors result in a call to `p_leave`. The content of the internal buffer is unpredictable in the event of an error.

★ FL_SENSE_REC**Sense current record number**

```
VOID fl_sense_rec(ULONG *prec);
```

Write, to **prec*, the record number of the record following the one which has last been read.

Writes zero if no records have been read since the receipt of an FL_REWIND message.

FL_DELREC**Delete a record**

```
VOID fl_delrec(UINT lsw, UINT msw);
VOID fl_delrec(ULONG recnum); (conceptual)
```

Delete record *recnum* (counting only records of types specified by *tlvfile.typmask*) by overwriting its record type with type 0. The *ULONG recnum* is actually passed in the message as two *UINT* parameters, *lsw* (least significant word) and *msw* (most significant word).

Calls *p_leave* on error, in which case the record may not have been deleted. It is the user's responsibility to determine whether the record has been deleted (for example, by testing the number of records).

FL_REPLACE**Replace a record**

```
VOID fl_replace(UINT lsw, UINT msw, OP_TLV_REC *prec);
VOID fl_replace(ULONG recnum, OP_TLV_REC *prec); (conceptual)
```

Replace a record by deleting record *recnum* (counting only records of types specified by *tlvfile.typmask*) and then appending the record specified by *prec*. The *ULONG recnum* is actually passed in the message as two *UINT* parameters, *lsw* (least significant word) and *msw* (most significant word).

Deletes the record by sending itself an FL_DELETE message and, if this is successful, appends the new record by sending itself an FL_WRITE_REC message.

Calls *p_leave* on error, in which case the original record may not have been deleted. If it has been deleted, the new record will either not have been appended or will have been marked as not being a valid record.

FL_READ_BY_TYPE**Read record of specific type(s)**

```
INT fl_read_by_type(UINT type);
```

Search forwards from the current file position and read into the internal buffer the first record whose type is one of those specified by the bitmask in *type* irrespective of the types specified by *tlvfile.typmask*.

Returns the record type of the record or, if the end of the file is reached before finding a record of a matching type, it returns *E_FILE_EOF*.

Calls *p_leave* for all other errors.

This method should be used with caution. If it skips records that would normally be read (because they are included in *tlvfile.typmask*) it may result in *tlvfile.pos* containing an incorrect value. If there is any doubt, this method should always be followed by the sending of an FL_REWIND message.

TLVDATA

TLVDATA
tlv cl_tlv changed index tmask name
td_open td_save td_changed td_load_item td_save_item td_reset td_set_file td_set_item td_sense_item

The TLVDATA abstract class provides the basic mechanisms for manipulating data that is stored in a series of records (each with a different record type) in a TLV file.

Class definition

Defined in sub-category file *tlvfile.cl* (generated header file *tlvfile.g*).

```
CLASS tlvdata root
Data which is loaded and saved to a tlvfile
{
  ADD td_open          Open file or revert to file
  ADD td_save          Save modifications to file
  ADD td_changed       Return TRUE if data changed since load
  ADD td_load_item     Load an item
  ADD td_save_item     Save an item
  ADD td_reset=p_dummy Clear data structures
  DEFER td_set_file     Set the file characteristics
  DEFER td_set_item    Set an item
  DEFER td_sense_item  Sense an item
  TYPES
  {
    typedef struct
    {
      OP_TLVFILE tlvfile;      File signature and mask
      TEXT ext[6];             Default extension
    } PR_TLVDATA_CHARS;
  }
  PROPERTY 1
  {
    PR_TLVFILE *tlv;           Handle to tlvfile
    UWORD cl_tlv;              Clean id for tlv file
    UWORD changed;             TRUE if data changed since load
    WORD index;                Index for load and save
    WORD tmask;                Valid record mask
    TEXT name[P_FNAME_SIZE];   Parameter file name
  }
}
```

Property

tlv	the handle of a temporary instance of the TLVFILE class. It should not be accessed by a subclass.
cl_tlv	the cleanup id for the TLVFILE instance, used for roll-back on error. It should not be accessed by a subclass.

changed	a flag indicating that the data has changed since a previous load or save. A subclass must set this to <code>TRUE</code> to signal that changed data requires saving. A subclass is not expected to set a <code>FALSE</code> value.
index	the type of the last record for which data has been sensed by the <code>td_save_item</code> method. It should not be accessed by a subclass.
tmask	the mask of valid record types. It should not be accessed by a subclass.
name	the full file specification of the file last used in the <code>td_open</code> method. It should not be accessed by a subclass.

TLVDATA methods

TD_OPEN

Open and read file

```
INT td_open(TEXT *name);
```

Open a TLV file, read all its records into memory, overwriting any existing data, and close it again.

Sends itself a `TD_RESET` message and then opens, reads and closes the file specified by `name`. If `name` is `NULL`, the file opened by a previous `td_open` or `td_save` is reopened, otherwise `name` should point to a string containing the name of the file to open.

Before opening the file, a `TD_SET_FILE` message is sent to determine the appropriate file characteristics and extension. If not `NULL`, the passed `name` is parsed (the related name being the file extension resulting from the `TD_SET_FILE` message) into the `tlvdata.name` buffer.

An instance of `TLVFILE` is created and used to open the file and read each record in turn. As each record is read, a `TD_LOAD_ITEM` message is sent, to store the record's content. When all the records have been read the `TLVFILE` instance is destroyed (which automatically closes the file). On successful conclusion the value of `tlvdata.changed` is set to `FALSE`.

Returns zero if successful, or `E_FILE_NXIST` if the specified file does not exist.

All other errors result in `p_leave` being called.

TD_SAVE

Save data to file

```
VOID td_save(TEXT *name);
```

Write the current data to the TLV file specified by `name`, replacing any existing file. If `name` is `NULL`, the file opened by a previous `td_open` or `td_save` is reopened, otherwise `name` should point to a string containing the name of the file to open.

Before opening the file, a `TD_SET_FILE` message is sent to determine the appropriate file characteristics and extension. If not `NULL`, the passed `name` is parsed (the related name being the file extension resulting from the `TD_SET_FILE` message) into the `tlvdata.name` buffer.

An instance of `TLVFILE` is created and used to open the file (to replace any existing file) and write the records. The data, length and type of each record is determined by sending a `TD_SAVE_ITEM` message. When this message returns zero, indicating that there are no further records, the `TLVFILE` instance is destroyed (which automatically closes the file). On successful conclusion the value of `tlvdata.changed` is set to `FALSE`.

Any error results in `p_leave` being called.

★ TD_CHANGED

Check if changed

```
INT td_changed(VOID);
```

Return `TRUE` if the data has been changed since a load or a save.

TD_LOAD_ITEM **Process a record read from a file**

```
VOID td_load_item(INT type, UBYTE *buf, UINT len);
```

Process the record by sending a TD_SET_ITEM message.

TD_SAVE_ITEM **Get a record to be saved**

```
INT td_save_item(UBYTE **pbuf, UWORD *plen);
```

Determine the type of the next record to be saved and send a TD_SENSE_ITEM message to sense its data.

The method uses `tlvdata.index` and `tlvdata.tmask` to identify the record type.

Returns the record type, or zero if there are no further records to be saved.

★ TD_RESET **Reset all data**

```
VOID td_reset(VOID);
```

The supplied method does nothing. It is expected to be subclassed to perform any appropriate reset action.

Deferred TLVDATA methods**TD_SET_FILE** **Set the TLV file characteristics**

```
VOID td_set_file(PR_TLVDATA_CHARS *pfc);
```

Write the appropriate TLV file signature header (including the mask of valid record types) and file extension to the `PR_TLVDATA_CHARS` struct pointed to by `pfc`.

On entry, `pfc->ext[0]` contains the character '.' and `pfc->tlvfile.fsig.offset` is already set to `sizeof(OP_BFILE_FSIG)`. All other bytes are set to zero.

TD_SET_ITEM **Set in-memory data for a record**

```
VOID td_set_item(INT type, VOID *buf, UINT len);
```

Store, in memory, the data for a record of type `type` and of length `len`, pointed to by `buf`.

TD_SENSE_ITEM **Sense in-memory data for a record**

```
INT td_sense_item(INT type, VOID **pbuf);
```

Write to `*pbuf` a pointer to the data for a record of type `type`.

Returns the length of the data.

SERFILE

TLVDATA	SERFILE
tlv	serial
cl_tlv	modem
changed	lnkdvr
index	serdvr
tmask	file
name	
td_open	td_reset
td_save	td_set_file
td_changed	td_set_item
td_load_item	td_sense_item
td_save_item	
~~td_reset~~	
~~td_set_file~~	
~~td_set_item~~	
~~td_sense_item~~	

The SERFILE class provides the methods for manipulating serial port parameter data saved in a *.trm* TLV file.

Class definition

Defined in sub-category file *tlvfile.cl* (generated header file *tlvfile.g*).

```

CLASS serfile tlvdata
{
  REPLACE td_reset          Set defaults
  REPLACE td_set_file       Set the file characteristics
  REPLACE td_set_item       Set an item
  REPLACE td_sense_item     Sense an item
  CONSTANTS
  {
    TE_MASK_SERIAL          0x0001
    TE_MASK_MODEM           0x0002
    TE_MASK_FILE            0x0004
    TY_SERFILE_SERIAL       1
    OBSOLETE_SERFILE_MODEM  2
    TY_SERFILE_FILE         3
    TY_SERFILE_NEW_MODEM    4
    TY_SERFILE_SERDVR       5
    TY_SERFILE_LNKDVR       6
    TE_XMDM_NONE            3
    TE_DIAL_PULSE           1
    TE_DIAL_TONE            2
    TE_MODEM_300            0x01
    TE_MODEM_1200           0x02
    TE_MODEM_2400           0x03
    TE_MODEM_4800           0x04
    TE_MODEM_9600           0x05
    TE_MODEM_19200          0x06
    MAX_DEVICE_NAME         10      ! TTY.AS5:A plus zero terminator
    MAX_MODEM_CMD           40
  }
}

```

```
TYPES
{
    typedef struct
    {
        ! Serial port
        P_SRCHAR ch;                Serial port characteristics
        TEXT port[P_MAXDEVNAME+2];  Serial port to use
    } PF_SERIAL;

    typedef struct
    {
        ! New Modem driver
        P_MDMCHR mch;
        UBYTE phone[25];            Phone number
        UBYTE auto_dial;            TRUE if modem to auto dial on connection
        UBYTE mdmdvr[MAX_DEVICE_NAME]; Modem driver to use (if any)
        UBYTE mdmcmd[MAX_MODEM_CMD]; Max extra configuration for modem
        UBYTE spare[16];
    } PF_MODEM;

    typedef struct
    {
        ! File transfer
        UWORD protocol;            File transfer protocol
    } PF_FILE;

    typedef struct
    {
        ! New Serial port
        P_SRCHAR ch;                Serial port characteristics
        UBYTE serdvr[MAX_DEVICE_NAME]; Serial driver to use
        UBYTE spare[16];
    } PF_SERDVR;

    typedef struct
    {
        ! Link drivers
        UBYTE masdvr[MAX_DEVICE_NAME]; Media access driver to use
        UBYTE lnkdvr[MAX_DEVICE_NAME]; Link driver to use
        UBYTE spare[16];
    } PF_LNKDVR;
}

PROPERTY
{
    PF_SERIAL serial;
    PF_MODEM modem;
    PF_FILE file;
    PF_LNKDVR lnkdvr;
    PF_SERDVR serdvr;
}
}
```

Property

Each item of property represents the data that may be stored in a record of a serial parameter *.trm* file.

serfile.serial	the data for a serial port record, of record type TY_SERFILE_SERIAL.
serfile.modem	the data for a modem driver record, of record type TY_SERFILE_NEW_MODEM.
serfile.file	the data for a file transfer record, of record type TY_SERFILE_FILE.
serfile.lnkdvr	the data for a link and media access driver record, of record type TY_SERFILE_LNKDVR.
serfile.serdvr	the data for an extended format serial driver record, of record type TY_SERFILE_SERDVR.

SERFILE methods

★ TD_RESET

Reset all data

```
VOID td_reset(VOID);
```

Set the serial data to its default values and then set `tlvdata.changed` to `TRUE`.

Sets the property data as follows:

```
serfile.serial
    ch.hand      P_OBEY_XOFF|P_SEND_XOFF|P_IGN_CTS
    ch.frame     P_DATA_8
    ch.tbaud     P_BAUD_9600
    ch.rbaud     P_BAUD_9600
    ch.xon       0x11 (DC1)
    ch.xoff      0x13 (DC3)
    ch.flags     P_IGNORE_PARITY
    port         "TTY:A"

serfile.modem
    mch.supported P_SRINQ_300|P_SRINQ_1200|P_SRINQ_2400|P_SRINQ_4800|P_SRINQ_9600|P_SR
    mch.baudrate  INQ_19200
    mch.chand     P_BAUD_2400
    mch.options   P_OBEY_DSR|P_FAIL_DSR|P_OBEY_DCD|P_FAIL_DCD|P_OBEY_XOFF|P_SEND_XOFF
    P_MDM_NO_MODULATION

serfile.file
    protocol     TE_XMDM_NONE

serfile.serdrv
    ch.hand      P_IGN_CTS
    ch.frame     P_DATA_8
    ch.tbaud     P_BAUD_19200
    ch.rbaud     P_BAUD_19200
    serdrv       "TTY:A"

serfile.lnkdrv
    masdrv       "MAS:"
    lnkdrv       "LLC:"
```

All fields not explicitly mentioned above are zero-filled.

★ TD_SET_FILE

Set the serial file characteristics

```
VOID td_set_file(PR_TLVDATA_CHARS *pfc);
```

Set the serial file signature, extension and record type mask in the `PR_TLVDATA_CHARS` struct pointed to by `pfc`.

The file signature is set to "TRM FILE" (with the remainder of the 16 bytes zero-filled) and the file name extension to ".TRM". The type mask is set for record types `TY_SERFILE_SERIAL`, `TY_SERFILE_NEW_MODEM`, `TY_SERFILE_FILE`, `TY_SERFILE_SERDRV` and `TY_SERFILE_LNKDRV`.

★ TD_SET_ITEM

Set in-memory data for a record

```
VOID td_set_item(INT type, VOID *buf);
```

Set the contents one of the five items of property corresponding to the record of type `type`, from the contents of `*buf`, where `type` is one of `TY_SERFILE_SERIAL`, `TY_SERFILE_NEW_MODEM`, `TY_SERFILE_FILE`, `TY_SERFILE_SERDRV` or `TY_SERFILE_LNKDRV` and `buf` correspondingly points to a `PF_SERIAL`, `PF_MODEM`, `PF_FILE`, `PF_SERDRV` or `PF_LNKDRV` struct.

Sets `tlvdata.changed` to `TRUE`.

★ TD_SENSE_ITEM Sense in-memory data for a record

```
INT td_sense_item(INT type, VOID **pbuf);
```

Write to *pbuf the address of one of the five items of property corresponding to the record of type type, where type is one of TY_SERFILE_SERIAL, TY_SERFILE_NEW_MODEM, TY_SERFILE_FILE, TY_SERFILE_SERDVR or TY_SERFILE_LNKDVR. The address written to *pbuf is a corresponding pointer to a PF_SERIAL, PF_MODEM, PF_FILE, PF_SERDVR or PF_LNKDVR struct.

Returns the length of the record data.

CHAPTER 9

THE CLEANUP CLASS

VAROOT		VAFIX	VAFLAT	CLEANUP
nrec	key	rlen	gran nspc base	level
~~destroy~~	~~va_replace~~	va_replace	va_init	destroy
va_count		va_copy	va_compress	cl_init
va_delete	~~va_copy~~	va_reclen	va_deletem	cl_add
va_sort	~~va_reclen~~	va_swap	va_insertm	cl_remove
va_key	~~va_swap~~		va_capacity	cl_clean_item
va_findisq	~~va_init~~		va_prec	cl_clean_level
va_insertisq	~~va_deletem~~		va_pbuf	cl_set_level
va_append	~~va_insertm~~			
va_insert	~~va_prec~~			
va_search	~~va_pbuf~~			
va_compare	~~va_capacity~~			
va_reset	~~va_compress~~			
va_test				

The CLEANUP class supplies one of the main mechanisms by which resources may be released following an error condition.

A typical use is in a case where a program allocates, say, a sequence of cells which must either exist as a whole or not at all. If, during the allocate sequence, one of the later allocations fails, the previously allocated cells must be freed.

The recovery process can be simplified if each cell is placed in a cleanup list as it is allocated. Once the whole sequence of allocations is complete the items may be removed from the list. If, however, there is a failure in one of the later stages, the previously allocated cells can be freed by the sending of, for example, a single CL_CLEAN_LEVEL message. Note that the process of adding an item to the cleanup list is so arranged that the addition itself can never fail due to shortage of memory.

Normally, an instance of the CLEANUP class is created as a component of the application manager (see the *Application Manager* chapter of this manual). In this case the cleaning up of partially complete allocations is generally handled automatically whenever an error occurs and the programmer's responsibility is reduced to adding items to, and removing items from, the cleanup list at the appropriate times.

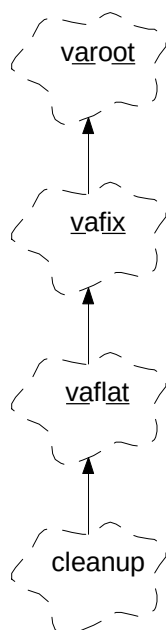
Because the addition and removal of items from a cleanup list that is a component of the application manager is so common, a set of convenience functions are provided. These are described in a separate section at the end of this chapter.

Precursors

The reader is assumed to understand:

- variable arrays of fixed length records
- the p_enter and p_leave error handling services

Class diagram



Class definition

Defined in sub-category file *appman.cl* (generated header file *appman.g*).

```
CLASS cleanup vaflat
{
  REPLACE destroy      Destroy all items in cleanup list
  ADD cl_init          Initialise cleanup table
  ADD cl_add            Add an item for cleanup
  ADD cl_remove        Remove an item (without cleanup)
  ADD cl_clean_item     Clean up an item
  ADD cl_clean_level   Clean up all items at current level
  ADD cl_set_level      Set the cleanup level

  CONSTANTS
  {
    TY_CLEANUP_DYL -5      A dyl handle
    TY_CLEANUP_SHARED -4   A shared allocated cell
    TY_CLEANUP_VOID -3     Already cleaned up
    TY_CLEANUP_ALLOC -2    An allocated cell
    TY_CLEANUP_IOCHAN -1   An IO channel
    TY_CLEANUP_OBJECT 0    An object (O_DESTROY method number !!)
  }

  TYPES
  {
    typedef struct
    {
      BYTE type;          Type of resource
      UBYTE level;        Cleanup level
      HANDLE h;           Handle of resource
    } RC_CLEANUP;
    typedef struct
    {
      UWORD nref;
    } SHARED_ALLOC;
  }

  PROPERTY
  {
    UWORD level;          current cleanup level
  }
}
```

Property

<code>cleanup.level</code>	the current cleanup level; set by <code>cl_set_level</code> and used by <code>cl_clean_level</code>
----------------------------	---

CLEANUP Methods

J DESTROY

Destroy

```
VOID destroy(VOID);
```

Clean up all items in the cleanup list and supersend a DESTROY message.

CL_INIT

Initialise list

```
VOID cl_init(UINT num);
```

Initialise the cleanup list.

Sends itself a VA_INIT message to initialise the array for records of length `sizeof(RC_CLEANUP)`, with a granularity of `num`. Then sends itself a VA_CAPACITY message to set the capacity to `num` records and ensures that the array contains at least one empty cleanup slot (of type `TY_CLEANUP_VOID`).

CL_ADD

Add item

```
UINT cl_add(UINT type, HANDLE h);
```

Add an item to the cleanup list of type `type` and handle `h` to the cleanup list at the current cleanup level.

The possible values of `type` are:

<code>TY_CLEANUP_OBJECT</code>	<code>h</code> is the handle of an object, as returned by <code>p_new</code>
<code>TY_CLEANUP_IOCHAN</code>	<code>h</code> is the pointer to the channel control block, as set by <code>p_open</code>
<code>TY_CLEANUP_ALLOC</code>	<code>h</code> is the pointer to the allocated cell, as returned by <code>p_alloc</code>
<code>TY_CLEANUP_SHARED</code>	<code>h</code> is the pointer to a shared allocated cell, as returned by <code>p_alloc</code> (the first word of a shared allocated cell is assumed to contain a usage count)
<code>TY_CLEANUP_DYL</code>	<code>h</code> is the category handle of a loaded dynamic library, as set by <code>p_loadlib</code>

All other values of `type`, except for `TY_CLEANUP_VOID`, are assumed to relate to objects and behave in a way similar to `TY_CLEANUP_OBJECT`. In all such cases `h` is assumed to be the handle of the object as returned by `p_new`. When such an item is cleaned up, it is sent a message with message number equal to the value of `type`. The value `TY_CLEANUP_OBJECT (0)` is chosen specifically to correspond to a DESTROY message.

A subclasser who adds further types should respect the current scheme by using a negative number for each new type leaving positive numbers to represent object message numbers.

Returns an index number which identifies the newly added item in the cleanup list.

The `cl_add` method may call `p_leave(E_GEN_NOMEMORY)`, *but not until after the current item has been added to the cleanup list*. Thus an item cannot be 'lost' by a failure when adding it to the cleanup list.

J CL_REMOVE

Remove item

```
VOID cl_remove(UINT num);
```

Remove the item specified by `num` (as returned by an earlier `CL_ADD` message) from the cleanup list without cleaning up its associated resource(s).

The item is removed by setting its type to `TY_CLEANUP_VOID`. No memory is freed.

J CL_CLEAN_ITEM

Delete item

```
VOID cl_clean_item(UINT num);
```

Clean up the resource(s) associated with item `num` (as returned by an earlier `CL_ADD` message) and remove the item from the cleanup list.

The cleanup action depends on the type of the item as follows:

TY_CLEANUP_IOCHAN	close the channel, using <code>p_close</code>
TY_CLEANUP_ALLOC	free the allocated cell, using <code>p_free</code>
TY_CLEANUP_SHARED	decrement the usage count of the shared allocated cell and, if decremented to zero, free the cell
TY_CLEANUP_DYL	unload the dynamic library, using <code>p_unloadLib</code>

All other values are assumed to relate to an object and the object is sent a message with message number equal to the type. The special case of type `TY_CLEANUP_OBJECT` (0) corresponds to the sending of a `DESTROY` message.

No allocated memory associated with the cleanup list's array is freed, but the item's type is set to `TY_CLEANUP_VOID` so that it is available for re-use.

J CL_CLEAN_LEVEL Delete all items at current level

```
VOID cl_clean_level(VOID);
```

Clean up all items that have been added at the current cleanup level (by sending a series of `CL_CLEAN_ITEM` messages).

J CL_SET_LEVEL Set cleanup level

```
VOID cl_set_level(UINT level);
```

Set the current cleanup level, stored in `cleanup.level`, to `level`.

CLEANUP convenience functions

These convenience functions assume that an instance of the `CLEANUP` class has been created during the initialisation of an instance of the application manager, and that its handle is stored in the application manager's property `appman.clean`. An application should ensure that the application manager's property is accessible via the 'magic static' `w_am`.

cl_add Add an item

```
INT cl_add(INT type, VOID *p);
```

Add an item, with handle `p` and of the specified type, to the cleanup list. The value of `type` may be one of:

```
TY_CLEANUP_OBJECT  
TY_CLEANUP_IOCHAN  
TY_CLEANUP_ALLOC  
TY_CLEANUP_SHARED  
TY_CLEANUP_DYL
```

Calling this function is equivalent to:

```
p_send4(w_am->appman.clean, 0_CL_ADD, type, p);
```

Returns an index number which identifies the newly added item in the cleanup list.

The function may call `p_leave(E_GEN_NOMEMORY)`, *but not until after the item has been added to the cleanup list*.

cl_add_object Add an object

```
INT cl_add_object(VOID *p);
```

Add an object, with handle `p`, to the cleanup list.

Calling this function is equivalent to calling:

```
cl_add(TY_CLEANUP_OBJECT, p);
```

Returns an index number which identifies the newly added item in the cleanup list.

The function may call `p_leave(E_GEN_NOMEMORY)`, *but not until after the item has been added to the cleanup list.*

cl_add_iochan

Add an I/O channel

```
INT cl_add_iochan(VOID *p);
```

Add an I/O channel, with handle `p`, to the cleanup list.

Calling this function is equivalent to calling:

```
cl_add(TY_CLEANUP_IOCHAN, p);
```

Returns an index number which identifies the newly added item in the cleanup list.

The function may call `p_leave(E_GEN_NOMEMORY)`, *but not until after the item has been added to the cleanup list.*

cl_add_alloc

Add an allocated cell

```
INT cl_add_alloc(VOID *p);
```

Add an allocated heap cell, with handle `p`, to the cleanup list.

Calling this function is equivalent to calling:

```
cl_add(TY_CLEANUP_ALLOC, p);
```

Returns an index number which identifies the newly added item in the cleanup list.

The function may call `p_leave(E_GEN_NOMEMORY)`, *but not until after the item has been added to the cleanup list.*

cl_add_shared

Add a shared allocated cell

```
INT cl_add_shared(VOID *p);
```

Add a shared allocated heap cell, with handle `p`, to the cleanup list.

Calling this function is equivalent to calling:

```
cl_add(TY_CLEANUP_SHARED, p);
```

Returns an index number which identifies the newly added item in the cleanup list.

The function may call `p_leave(E_GEN_NOMEMORY)`, *but not until after the item has been added to the cleanup list.*

cl_add_dyl

Add a DYL

```
INT cl_add_dyl(VOID *p);
```

Add a DYL, with handle `p`, to the cleanup list.

Calling this function is equivalent to calling:

```
cl_add(TY_CLEANUP_DYL, p);
```

Returns an index number which identifies the newly added item in the cleanup list.

The function may call `p_leave(E_GEN_NOMEMORY)`, *but not until after the item has been added to the cleanup list.*

cl_remove**Remove an item**

```
VOID cl_remove(INT num);
```

Remove the item specified by `num` (as returned by an earlier `CL_ADD` message or a call to one of the `cl_add` convenience functions) from the cleanup list without cleaning up its associated resource(s).

The item is removed by setting its type to `TY_CLEANUP_VOID`. No memory is freed.

Calling this function is equivalent to:

```
p_send3(w_am->appman.clean, O_CL_REMOVE, num);
```

cl_clean_item**Delete an item**

```
VOID cl_clean_item(INT num);
```

Clean up the resource(s) associated with item `num` (as returned by an earlier `CL_ADD` message or a call to one of the `cl_add` convenience functions) and remove the item from the cleanup list.

No allocated memory associated with the cleanup list's array is freed but the item's type is set to `TY_CLEANUP_VOID` so that it is available for re-use.

Calling this function is equivalent to:

```
p_send3(w_am->appman.clean, O_CL_CLEAN_ITEM, num);
```

CHAPTER 10

THE APPMAN APPLICATION MANAGER CLASS

APPMAN
<code>clean</code> <code>system</code> <code>rcb</code> <code>srcb</code> <code>ipcs</code> <code>task</code> <code>stop</code> <code>nrid</code> <code>err</code> <code>spare1</code> <code>spare2</code>
<code>am_init</code> <code>am_start</code> <code>am_stop</code> <code>am_add_task</code> <code>am_wait</code> <code>am_load_resource</code> <code>am_load_res_buf</code> <code>am_rscname</code> <code>am_notify</code> <code>am_notifyerr</code> <code>am_cleanup</code> <code>am_onlyone</code> <code>am_finding</code> <code>am_change_pri</code>

The main function of the application manager class APPMAN is to provide an application's central logic for scheduling the processing of events which may derive from more than one source. As such, it is fundamental to the operation of a SIBO application, providing the basic support for a multi-threaded approach to the processing of events from different sources (such as keypresses, the receipt of data from a serial port and the expiry of timers).

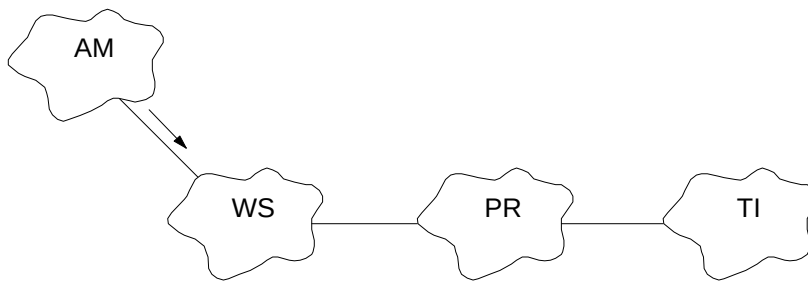
In addition, the application manager supplies some general utilities, including methods to access resources held in resource files, together with some basic error handling and notification services.

An application process almost invariably creates an instance of the application manager or, more commonly, an instance of a user interface subclass of the application manager (for example, HWIMMAN - see the *HWIM Reference* manual). This instance normally remains in existence for the lifetime of the application process.

The reserved static `w_am` is intended to be used to store the handle of an application's application manager, making its methods accessible from any part of the application code.

Each of the events that are scheduled by APPMAN is represented by an active object, i.e. an instance of a subclass of `ACTIVE`. The application manager maintains a queue, in priority order, of instances of active objects and the `am_start` method schedules processing between them in a non pre-emptive way.

For example, an application which is printing could have the following structure:



where the application manager (AM) holds a queue of three active objects, WS, PR and TI, representing:

- the window server (WS)
- a printer channel (e.g. a serial port) (PR)
- an asynchronous timer (TI)

In this example, printing is performed by making a write request on the printer active object and a time-out request on the timer active object. (Note that, in general, there will also be an outstanding event read request on the window server active object.)

The application manager waits for an event which, in this case, will be the completion of any one of the requests on the three active objects. When an event occurs, the application manager scans its active object queue to determine which active object has a completed request and is prepared to run. The application manager will then send an `AO_RUN` message to the appropriate active object.

If, in our example, the printer write request completes, the printer active object will be sent an `AO_RUN` message. The printer's `ao_run` method will typically cancel the timer's time-out request and then repeat its own write request and the timer's time-out request, to continue printing. Alternatively, if the time-out expires, the timer active object will be sent an `AO_RUN` message. The timer's `ao_run` method will abandon printing by cancelling the write request on the printer active object.

A window server read event may complete at any point in the printing process - for example, to redraw a window or to indicate loss of foreground. In this case the window server active object will receive an `AO_RUN` message and the processing of the window server event is automatically interleaved with the processing of the write and time-out events but note that the processing of a write or a timeout event cannot be *interrupted* to handle a window server event.

Active object priorities

The application manager's queue of active objects is maintained, and scanned, in priority order. The priority is a signed value, so that the default value of zero is in the middle of the range. The range of predefined priorities is given in the *Active Objects* chapter of this manual.

If more than one active object has generated an event, the first task in the queue is given absolute priority - a task at the end of the queue only runs when all earlier tasks are not prepared to run. It is fundamental to the scheduling process to note that:

- the events which signal the completion of requests do not necessarily occur in the order in which the requests were made
- the active objects are not necessarily given an opportunity to run in the order of completion of the corresponding requests - if more than one request has completed, the scheduling mechanism will give the object with the highest priority the first opportunity to run.

However, each active object which makes a request will, of course, eventually receive an invitation to run at some time following the completion of its request.

Once the application manager has sent an `AO_RUN` message to an active object, no other active object can be given an opportunity to run until the processing of the `AO_RUN` message is complete and the `ao_run` method has returned. The application manager has no means of preempting the current active object (contrast this with the EPOC operating system in which scheduling is preemptive). If the processing of an event takes an extended time to perform, all sources of events are blocked for that period of time and this may reduce the perceived quality of the application. In particular, the processing of user input (seen as an event from the window server) is delayed - the application temporarily *goes deaf*. A technique for coping with this situation is discussed in the *Idle Objects and the AIDLE Class* chapter of this manual.

Active object scheduling

APPMAN's active object scheduling is a complex process, requiring close cooperation between APPMAN and the active objects in its queue. During the process, APPMAN reads and modifies property elements of the active objects. It is, therefore, not possible to discuss the scheduling process without making some reference to the behaviour of active objects. Perhaps the clearest approach is to consider what qualifies an active object to be offered a chance to run.

The first requirement is that it must have made a request to be run, usually by execution of its `ao_queue` method. At this point it will have triggered a sequence which will eventually result in an event being detected by APPMAN. APPMAN detects an event by calling `p_iowait` which returns when `p_iosignal` is called.¹ The `ao_queue` method will normally trigger a `p_iosignal` by making an asynchronous request, during which the `active.stat` field is usually set to `E_FILE_PENDING` (but see the exception discussed below, under the heading *The ao_run return value*). The making of a request is indicated by the active object changing state, from *inactive* to *active* (the `active.isactive` field changes from `FALSE` to `TRUE`).

The later completion of the asynchronous request results in `active.stat` being set to a value other than `E_FILE_PENDING`.

APPMAN detects an event by the receipt of a signal on the I/O semaphore of its process. At this point the application manager scans, in priority order, all active objects in its queue. If, in the property of an active object, `active.isactive` is set to `TRUE`, the value of `active.stat` is examined. If this is set to any value other than `E_FILE_PENDING` the active object is assumed to be prepared to consume the event and is sent an `AO_RUN` message. Normally, the active object confirms that it has consumed the event (signal) by returning the value `RUN_ACTIVE_USED` (see below, under the heading *The ao_run return value*, for exceptions).

If an active object confirms that it has consumed the event, the application manager scheduling loop waits for the next event, otherwise it continues looking for an active object that can consume the event. The fatal condition, known as *stray signal death*, occurs if the application manager reaches the end of its list before any active object consumes the event. In this situation the application manager calls `p_panic(P_PANIC_APPMAN_1)`. (This panic has the value 143.)

(For further details of asynchronous processes, see the *Asynchronous Requests and Semaphores* chapter in the *PLIB Reference* manual.)

Note that APPMAN reads an active object's `active.isactive` and `active.stat` property fields for reasons of efficiency. It avoids the duplication of the tests of these fields in the `ao_run` method of each active object and, more importantly, executes more efficiently since messages are not sent to active objects that are not prepared to run.

The ao_run return value

Normally, an active object will represent a source of events of a single type. According to the above description of the scheduling mechanism, the active object will not be sent an `AO_RUN` message unless the corresponding asynchronous event has completed. In consequence, the `ao_run` method of such an active object can only ever return the value `RUN_ACTIVE_USED`.

For largely historical reasons an `ao_run` method may return `RUN_ACTIVE_UNUSED` to indicate that it has not consumed the event. This could, for example, be of use where a single active object is used to represent two or more related event sources of different types, for example, serial port reads and writes.

Such an active object would need to maintain a separate status word (in its property) for each type of asynchronous request, leaving `active.stat` with a permanent zero value. It would then be liable to receive an `AO_RUN` message at any time that `active.isactive` is `TRUE`, regardless of the completion status of any of its outstanding asynchronous requests. The `ao_run` method should only return `RUN_ACTIVE_UNUSED` if none of its outstanding requests have completed.

This technique, although relatively simple to implement, is inefficient if the application contains other active objects of equal or lower priority. In such a situation the active object will, in general, be sent a number of 'unnecessary' `AO_RUN` messages. From an architectural point of view, and in the interests of efficient execution, it is better to implement the handling of multiple event sources by using a separate active object for each event source. Each active object will then only be sent an `AO_RUN` message when its corresponding outstanding request has completed (and will always return the value `RUN_ACTIVE_USED`).

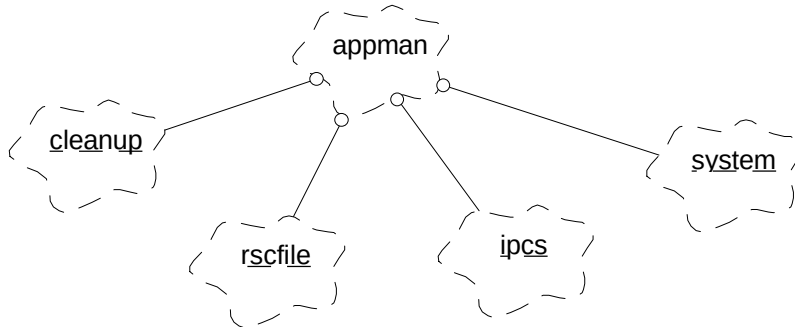
¹ The call to `p_iosignal` is thus the event source.

Precursors

The reader is assumed to understand:

- the PLIB/EPOC I/O system, waits, signals and semaphores.
- the requirements of an event-driven system.
- the `p_enter` and `p_leave` error handling services.

Class diagram



APPMAN may optionally reference (use) one or more of the CLEANUP, RSCFILE, IPCS and SYSTEM classes.

Class definition

Defined in sub-category file `appman.cl` (generated header file `appman.g`).

```

CLASS appman root
Application manager - schedules attached active objects
{
  ADD am_init           Initialise task queue
  ADD am_start          Start a scheduling loop
  ADD am_stop           Exit one level of the scheduling loop
  ADD am_add_task       Insert active object into task queue
  ADD am_wait=p_iowait  Wait for the next signal
  ADD am_load_resource  Load a resource file record into memory
  ADD am_load_res_buf   Load a resource file record into buf supplied
  ADD am_rscname        Supply a resource file name at initialisation
  ADD am_notify         Notify user
  ADD am_notifyerr      Notify user of error
  ADD am_clean_up       Clean all logged objects then do an abrun
  ADD am_onlyone        Called when the only one check fails
  ADD am_finding        Re-find the image if the pack is moved
  ADD am_change_pri     Change the priority of an active object
CONSTANTS
  {
    FLG_APPMAN_CLEAN    0x01    Create a cleanup list component
    FLG_APPMAN_SYSTEM   0x02    Create a system configuration component
    FLG_APPMAN_RSCFILE  0x04    Create a resource file component
    FLG_APPMAN_SRSCFILE 0x08    Create a system resource file component
    FLG_APPMAN_IPCS     0x10    Create an ipcs component
    FLG_APPMAN_ONLYONE  0x20    Fail if same process already exists
    FLG_APPMAN_NODBG    0x40    Don't grope for dbg.dyl if set
    RUN_ACTIVE_UNUSED   0       Signal not used
    RUN_ACTIVE_USED     1       Signal used
    ERR_APPMAN_APPL     -512    Base for application specific leaves
  }
PROPERTY 5
  {
    PR_CLEANUP *clean;    cleanup list for leaves
    PR_SYSTEM *system;    system configuration object
    PR_RSCFILE *rcb;      application resource file
    PR_RSCFILE *srcb;     system resource file
    PR_IPCS *ipcs;        ipcs object
    P_QUE task;           queue header
    UWORD stop;           start level counter
    WORD nrid;            context message rid for notify
    WORD err;             abrun error
    UBYTE *spare1;        Spare for future expansion.
    UBYTE *spare2;        Spare for future expansion.
  }
}

```

Property

<code>appman.clean</code>	Contains the handle of the created CLEANUP object if the <code>FLG_APPMAN_CLEAN</code> flag was specified to <code>am_init</code> . The CLEANUP object handle is used when an active object's <code>ao_run</code> method leaves with error. All applications will normally create a CLEANUP object. This field should be treated as read only by all objects.
<code>appman.system</code>	Contains the handle of the created SYSTEM configuration object if the <code>FLG_APPMAN_SYSTEM</code> flag was specified to <code>am_init</code> . This field should be treated as read only by all objects.
<code>appman.rcb</code>	Contains the handle of the created application resource file RSCFILE object if the <code>FLG_APPMAN_RSCFILE</code> flag was specified to <code>am_init</code> . This object is used in the <code>am_load_resource</code> and <code>am_load_res_buf</code> methods, provided the specified resource id is positive. This field should be treated as read only by all objects. Note that RSCFILE objects require the use of the application manager's CLEANUP object.
<code>appman.srcb</code>	Contains the handle of the created system resource file RSCFILE object if the <code>FLG_APPMAN_SRSCFILE</code> flag was specified to <code>am_init</code> . This object is used in <code>am_load_resource</code> and <code>am_load_res_buf</code> methods if the specified resource id is negative. This field should be treated as read only by all objects. Note that RSCFILE objects require the use of the application manager's CLEANUP object.
<code>appman.ipcs</code>	Contains the handle of the created IPCS object if the <code>FLG_APPMAN_IPCS</code> flag was specified to <code>am_init</code> . This field should be treated as read only by all objects.
<code>appman.task</code>	This is the head of the active object task queue. All active objects are inserted into this queue when they send the application manager an <code>AM_ADD_TASK</code> message. The queue is maintained in priority order. Note that if an active object wishes to change its priority it should send APPMAN an <code>AM_CHANGE_PRI</code> message; just changing the priority property field is not sufficient. This field should not be accessed by any subclass.
<code>appman.stop</code>	Maintains the current level of active object event scheduling, as set by the <code>am_start</code> and <code>am_stop</code> methods. This field should not be accessed by any subclass.
<code>appman.nrid</code>	This field is intended to be used by applications to store a resource id to be used in reporting errors. The idea is that this resource id changes as the execution of code progresses, the id providing a context of where the error(s) are occurring. APPMAN makes no use of this variable itself, but it is used by ACTIVE's <code>ao_abrun</code> method.
<code>appman.err</code>	Contains the error returned by the <code>ao_run</code> method of an object. The error number is placed there by the <code>am_clean_up</code> code before the <code>ao_abrun</code> method is called. This allows the error to be more accessible than if it were just passed as a parameter, and also reduces stack build up.
<code>appman.spare1</code> <code>appman.spare2</code>	Reserved for future expansion.

APPMAN Methods**AM_INIT****Initialise**

```
VOID am_init(UINT flags);
```

Initialise the task queue and create a series of objects as determined by `flags` which should contain a combination of the flag values listed below.

If an error is encountered during initialisation `p_leave` is called with the appropriate error. If the initialisation fails, an application must assume that none of the requested objects have been created. In particular, no resource files will have been opened and hence the application can only exit, without attempting to use any resource data.

The following descriptions of the various flag values make reference to a number of other object classes. For more information on any of these classes, see the appropriate section of this manual.

FLG_APPMAN_CLEAN	This flag causes an instance of the CLEANUP class to be created and initialised with a granularity of 8. This is used to lodge items that must be 'cleaned up' on error.
FLG_APPMAN_SYSTEM	This flag causes an instance of the SYSTEM class to be created and initialised (see the <i>System Services</i> chapter). This is used to obtain system-wide information.
FLG_APPMAN_RSCFILE	This flag causes an instance of the RSCFILE class to be created to provide access to the application resource file (see the <i>Resource Files</i> chapter). The name of the resource file is generated by the <code>am_rscname</code> method, which should be subclassed if a different name is required. Note that the RSCFILE class requires the presence of an instance of the CLEANUP class, so this flag must always be accompanied by FLG_APPMAN_CLEAN.
FLG_APPMAN_SRSCFILE	This flag causes an instance of the RSCFILE class to be created, to provide access to the system resource file (see the <i>Resource Files</i> chapter). Note that the RSCFILE class requires the presence of an instance of the CLEANUP class, so this flag must always be accompanied by FLG_APPMAN_CLEAN.
FLG_APPMAN_IPCS	This flag causes an instance of the IPCS class to be created and initialised with a maximum message size of 8 bytes, in a queue of length 4 (see the <i>Inter-process Communication</i> chapter). If this is insufficient then the application should not use this flag, but should explicitly create its own IPCS object.
FLG_APPMAN_ONLYONE	This flag should be set if there must be only one process of this type running at any one time. It is, for example, set for Alarms, Link and the System process. If this flag is set and another process exists with the same name as the one now being run, APPMAN sends itself an AM_ONLYONE message, passing the process id of the other process.
FLG_APPMAN_NODBG	In the absence of Psion's internal test and debug library, <i>dbg.dyl</i> , this flag has no effect. If this flag is clear and <i>dbg.dyl</i> is present in the appropriate directory, a debug object is created to run test procedures and display various items of debug information for the application.

★ AM_WAIT

Wait on I/O semaphore

```
VOID am_wait(VOID)
```

Waits on the I/O semaphore by calling `p_iowait`.

When an event occurs, it signals the I/O semaphore which causes `p_iowait`, and hence `am_wait`, to return.

AM_START

Start scheduler

```
VOID am_start(VOID)
```

Start a new level of the application manager active object event scheduler. This method normally does not return until the application manager receives an AM_STOP message.

The code of this method is presented below, since it is crucial to the understanding of the application manager's active object scheduling mechanism.

```

LOCAL_C RunTask(FAST PR_ACTIVE *htask)
{
    INT ret;

    htask->active.isactive=FALSE;
    ret=p_send2(htask,O_AO_RUN);
    if (ret==RUN_ACTIVE_UNUSED)
        htask->active.isactive=TRUE;
    return(ret);
}

LOCAL_C RunCleanupAbrun(PR_APPMAN *self,INT err,VOID *htask)
/*
Enterable shell
*/
{
    p_send4(self,O_AM_CLEAN_UP,err,htask);
    return(0);
}

METHOD VOID appman_am_start(PR_APPMAN *self)
/*
Start the basic loop to get a message from the server.
May be called recursively for modal interaction.
*/
{
    INT stop,ret,abret;
    FAST P_QUE *pt;
    FAST PR_ACTIVE *htask;

    stop=(++self->appman.stop);
    if (self->appman.clean)
        p_send3(self->appman.clean,O_CL_SET_LEVEL,stop);
    do
    {
        p_send2(self,O_AM_WAIT); /* wait for next event */
        for (pt=self->appman.task.next;;pt=pt->next)
        { /* find a task to run */
            if (pt==&self->appman.task)
                p_panic(P_PANIC_P_APPMAN_1); /* stray signal */
            htask=(PR_ACTIVE *)(((UBYTE *)pt)-sizeof(PR_ROOT));
            if (htask->active.isactive && htask->active.stat!=E_FILE_PENDING)
            {
                abret=0;
                if ((ret=p_enter2(RunTask,htask))<0) /* p_leave(err) called */
                {
                    abret=p_enter4(RunCleanupAbrun,self,ret,htask);
                    if (abret)
                        self->appman.stop--;
                }
                if (ret!=RUN_ACTIVE_UNUSED)
                    break;
            }
        }
    } while (self->appman.stop==stop);
    if (self->appman.clean)
    {
        p_send2(self->appman.clean,O_CL_CLEAN_LEVEL);
        p_send3(self->appman.clean,O_CL_SET_LEVEL,stop-1);
    }
    if (abret)
        p_leave(abret); /* allow +ve 'errors' */
}

```

On entry, `appman.stop` is incremented. Provided `appman.clean` is non-zero, the CLEANUP object is sent a `CL_SET_LEVEL` message to set its level to the new value of `appman.stop`.

The scheduler waits for an event by sending itself an `AM_WAIT` message. On return, all objects that are currently active (`active.isactive` set to `TRUE`) have their completion status words (`active.stat`) checked. If the completion status word is not `E_FILE_PENDING` the object is sent an `AO_RUN` message.

Assuming that there are no errors, if the object does not consume the event it must return `RUN_ACTIVE_UNUSED`, otherwise it returns any other value, normally returning `RUN_ACTIVE_USED`.

If the list of active objects is exhausted without the event being consumed, the scheduling loop calls `p_panic(P_PANIC_APPMAN_1)`, indicating the *stray signal death* condition.

Before sending the `AO_RUN` message, the application manager sets the active object's `active.isactive` to `FALSE`. If the object returns `RUN_ACTIVE_UNUSED`, `active.isactive` is set back to `TRUE` since the object must still be left in the active state if it does not consume the event. If the active object leaves with an error, as described below, the `TRUE` value is not written to `active.isactive`. (Note that changing the value of `active.isactive` is a code-saving service, avoiding the duplication of code in each active object.)

Any error in the `ao_run` method is expected to result in `p_leave` being called. The `AO_RUN` message is sent under the protection of a `p_enter` which catches any `p_leave` error calls. If an active object calls `p_leave` within its `ao_run` method the application manager will receive an error (negative) return value and will then send an `AM_CLEAN_UP` message, passing the error that was detected and the handle of the active object that called `p_leave`.

The `AM_CLEAN_UP` message is also sent under the protection of a `p_enter` and any non-zero return value (representing a `p_leave` exception in the error handling code) is stored for later use. Note that this will cause the current level of event scheduling to terminate, with the error being propagated to the previous level. The technique of calling `p_leave` within the error handling code should therefore only be used with extreme caution.

Note that an `ao_run` method is free to terminate its processing prematurely by calling `p_leave` with a zero or positive argument - a preferred form of the call is `p_leave(RUN_ACTIVE_USED)`. Such termination will not trigger the error reporting and recovery mechanism and may be considered equivalent to a normal termination that returns `RUN_ACTIVE_USED`.

Once an object consumes the signal, by returning a value other than `RUN_ACTIVE_UNUSED` from its `ao_run` method, no more objects are polled. At this point the `am_start` method normally loops back to send itself another `AM_WAIT` message to wait for the next event.

The exceptions to this are:

- if an active object has sent an `AM_STOP` message in its `ao_run` method
- if a non-zero return value resulted from the `AM_CLEAN_UP` message.

In either case `appman.stop` will have been decremented. On completion of the processing of the current event, the `AM_START` method returns, exiting one level of scheduling.

Before returning, the application manager's `CLEANUP` object (if it exists) is sent a `CL_CLEAN_LEVEL` message, to discard all items still in the cleanup list at the current level. It is then sent a `CL_SET_LEVEL` message to adjust it to the new (lower) scheduling level. If a non-zero value resulted from the `AM_CLEAN_UP` message, `p_leave` is called, passing this value, to propagate the exception generated in the error handling code to the previous level of scheduling.

The application manager active object event scheduler is re-entrant. Thus the `ao_run` method of an active object can send the application manager an `AM_START` message to enter a further level of scheduling. Normally, the active object which sends the `AM_START` message will, at that time, have `active.isactive` set to `FALSE` (by the application manager, before it sends the `AO_RUN` message) and will therefore not receive any further `AO_RUN` messages until an `AM_STOP` message is sent. Events occurring under other active objects in the application manager's queue will, however, continue to be processed as normal. An important example of such use is when a modal dialog box is being run from a menu selection.

There is a great temptation to use this technique to implement any synchronous sequence of actions by means of an active object whose initialisation method, say, sends an `AO_QUEUE` message and then sends the application manager an `AM_START` message. This technique should be used with care, particularly when recovery from an error involves items on the cleanup list.

Because of the different level, the items that are cleaned will be different for an error that occurs before an `AM_START` message (for example, during an initialisation phase) from the items that are cleaned up after (say, within an `ao_run` method). It may be advisable to transfer any part of the initialisation that can fail into an `ao_run` method and execute it under the control of a state variable in the object's property, the first time that the `ao_run` method is called. This also resolves any problem as to whether the error recovery code should or should not send an `AM_STOP` message. The following general code briefly illustrates the principle of implementing such an active object sequencer:

```

GLREF_D PR_APPMAN *w_am;

VOID sequence_ao_init(PR_SEQUENCE *self)
{
    self->active.priority=PRIORITY_ACTIVE_COMPUTE;
    p_send3(w_am,0_AM_ADD_TASK,self);
    self->active.isactive=TRUE;
    p_iosignal();
    p_send2(w_am,0_AM_START);
}

sequence_ao_run(PR_SEQUENCE *self)
{
    switch (self->sequence.state_variable)
    {
        case 0:
            /* initialise */
            break;
        case 1:
            /* action 1 */
            break;
        case 2:
            ...
            ...
        case 5:
            p_send2(w_am,0_AM_STOP);
            return(RUN_ACTIVE_USED);
    }
    self->sequence.state_variable+=1;
    self->active.isactive=TRUE;
    p_iosignal();
    return(RUN_ACTIVE_USED);
}

```

★ AM_STOP

Stop scheduler

```
VOID am_stop(VOID)
```

Stop the current level of the active object event scheduling loop, by decrementing `appman.stop`. This causes the most nested `am_start` to return after handling of the current event is complete.

★ AM_ADD_TASK

Add a task

```
VOID am_add_task(PR_ACTIVE *hand);
```

Add an initialised active object to the task queue, in priority order, as determined by the active object's `active.priority` field.

The item is added to the list immediately following all existing items with the same (or higher) priority.

AM_LOAD_RESOURCE

Load a resource

```
INT am_load_resource(INT resid, UBYTE **ppdata);
```

Allocate a buffer and load into it the resource with id `resid` from the appropriate resource file.

A negative `resid` indicates that the resource is to be found in the system resource file, with an id equal to the absolute value of `resid`. Otherwise it is assumed that the resource is to be found in the application resource file.

If you wish to access the system resource file you *must* have passed the `FLG_APPMAN_SRSCFILE` flag to the `am_init` method. It is recommended, but not strictly essential (because the `am_load_resource` method will search for and open the application resource file if it is not already open) that you pass the `FLG_APPMAN_RSCFILE` flag to the `am_init` method if you wish to access the application resource file.

If the appropriate `RSCFILE` object exists, the resource is loaded by sending an `RS_READ` message to the appropriate `RSCFILE` object, under the protection of `p_enter`. If the loading of the resource fails due to out of memory then `p_leave(E_GEN_NOMEMORY)` is called.

Any other error when attempting to read an application resource, including the absence of the application resource RSCFILE object, is assumed to be due to the removal of the application resource file. (It is assumed that only the application resource file can be removed since the system resource file is in the ROM.) If the application manager's application RSCFILE component exists, it is destroyed. An attempt is then made to locate the resource file by sending AM_FINDIMG and AM_RSCNAME messages. If this is successful the RSCFILE object is recreated and the file reopened - either of which could fail, calling p_leave(E_GEN_NOMEMORY) - otherwise the method calls p_leave(E_FILE_NXIST). Following this, the resource is loaded by sending the appropriate RSCFILE object an RS_READ message, which may fail - typically by calling p_leave(E_GEN_NOMEMORY).

The method returns the size of the loaded resource, as returned by the RS_READ message.

AM_LOAD_RES_BUF

Load a resource to a buffer

```
INT am_load_res_buf(INT resid, UBYTE *pbuf);
```

Load into the buffer pointed to by pbuf the resource with id resid from the appropriate resource file.

A negative resid indicates that the resource is to be found in the system resource file, with an id equal to the absolute value of resid. Otherwise it is assumed that the resource is to be found in the application resource file.

If you wish to access the system resource file you *must* have passed the FLG_APPMAN_SRSCFILE flag to the am_init method. It is recommended, but not strictly essential (because the am_load_resource method will search for and open the application resource file if it is not already open) that you pass the FLG_APPMAN_RSCFILE flag to the am_init method if you wish to access the application resource file.

If the appropriate RSCFILE object exists, the resource is loaded by sending an RS_READ message to the appropriate RSCFILE object, under the protection of p_enter. If the loading of the resource fails due to out of memory then p_leave(E_GEN_NOMEMORY) is called.

Any other error when attempting to read an application resource, including the absence of the application resource RSCFILE object, is assumed to be due to the removal of the application resource file. (It is assumed that only the application resource file can be removed since the system resource file is in the ROM.) If the application manager's application RSCFILE component exists, it is destroyed. An attempt is then made to locate the resource file by sending AM_FINDIMG and AM_RSCNAME messages. If this is successful the RSCFILE object is recreated and the file reopened - either of which could fail, calling p_leave(E_GEN_NOMEMORY) - otherwise the method calls p_leave(E_FILE_NXIST). Following this, the resource is loaded by sending the appropriate RSCFILE object an RS_READ message, which may fail - typically by calling p_leave(E_GEN_NOMEMORY).

The method returns the size of the loaded resource, as returned by the RS_READ_BUF message.

AM_RSCNAME

Generate resource file name

```
VOID am_rscname(UBYTE *pname);
```

Write to the buffer at *pname (which must be at least P_FNAME_SIZE bytes long) the default full file specification (see the *Files* chapter of the *PLIB Reference* manual) of the application resource file.

The name is generated from the application's start-up full file specification, pointed to by the *magic static* DatCommandPtr.

The resource file is assumed to be built into the image file, so that the full file specification is identical to that of the image file.

★ AM_NOTIFY

Display notifier

```
VOID am_notify(UINT mess1, UINT mess2, UWORD *pbut);
```

Call the p_notify service with text loaded from resource files.

Up to two text messages are specified by the resource ids mess1 and mess2. If pbut is NULL, the single default button 'CONTINUE' (or the non-English equivalent) will be displayed. Otherwise, pbut is assumed to point to an array of three resource ids for the three notifier buttons.

All resource ids follow the resource id rules as specified in the description of the am_load_resource method. If any id is NULL then no text is loaded for that id.

Once the resource strings are loaded the `p_notify` service is invoked. The allocated space for the resource strings is freed after use.

This method will not fail due to lack of memory. If there is not enough memory available to load any of the specified resources, the corresponding part of the notification text is not displayed.

★ AM_NOTIFYERR

Notify an error

```
VOID am_notifyerr(INT err, UINT mess1, UWORD *pbut);
```

Call the `p_notifyerr` service, with text loaded from resource files.

A first line text message is specified by the resource id `mess1`. A second line contains a description of the error, as generated by `p_errs(err)`. If `pbut` is `NULL`, the single default button 'CONTINUE' (or the non-English equivalent) will be displayed. Otherwise, `pbut` is assumed to point to an array of three resource ids for the three notifier buttons.

All resource ids follow the resource id rules as specified in the description of the `am_load_resource` method. If any id is `NULL` then no text is loaded for that id.

Once the resource strings are loaded the `p_notifyerr` service is invoked, which converts the error number `err` into the second line text message. The allocated space for the resource strings is freed after use.

This method will not fail due to lack of memory. If there is not enough memory available to load any of the specified resources, the corresponding part of the notification text is not displayed.

AM_CLEAN_UP

Clean up resources and report an error

```
VOID am_clean_up(INT err, UBYTE *htask);
```

Provide standard error recovery and reporting for the active object event scheduler.

This method is called from the `am_start` event scheduler if an active object's `ao_run` method calls `p_leave(error)`. The handle of the active object is in `htask` and `err` is the error number passed to `p_leave`.

The value of `err` is copied to `appman.err` and if there is a `CLEANUP` object it is sent a `CL_CLEAN_LEVEL` message to tidy up all resources added to the cleanup list at this level of event scheduling.

If `htask` is not `NULL`, an `AO_ABRUN` message is sent to that object. The `ao_abrun` method may call `p_leave`, in which case the error will be caught in the `am_start` method. It will cause the current level event scheduling to terminate, the `p_leave` error being propagated to the next level of scheduling.

This method is supplied in order to facilitate the customising of all, or a particular set of, errors.

Note that the active object which generated the error in its `ao_run` method is sent an `AO_ABRUN` message *after* the sending of the `CL_CLEAN_LEVEL` message. This means that (unless the `am_cleanup` method is subclassed) the active object must not itself be in the cleanup list at the current level, otherwise it will be destroyed before the `AO_ABRUN` message is sent. It is likely, in any practical case, that an active object that has been placed in the cleanup list will have been removed before it receives its first `AO_RUN` message.

In general, the active object will only be placed in the cleanup list temporarily while other objects are being built and resources acquired; in this state of construction, it is unlikely that an application would "activate" the active object and risk receiving an `AO_ABRUN` message.

AM_FINDIMG

Find application image file

```
INT am_findimg(VOID);
```

Relocate the SSD from which the application was run.

It uses the *magic static* `DatCommandPtr`, assuming that it points to the current full file specification of the application's *.img* (or *.app*) file. It looks in all available SSD drives (A and B and, if they exist, C and D). If it finds a file with the same name in the directory specified by `DatCommandPtr` it patches the data at `DatCommandPtr` to reflect the new path and returns zero. If no such file can be found `E_FILE_NXIST` (the return value from a `p_info` call) is returned.

Typically this is called when the application wishes to access some information from the SSD from which it was run, but finds that the SSD is no longer in that drive. It is used in this way by the `am_load_resource` and `am_load_res_buf` methods.

AM_ONLYONE **Ensure only one copy running**

```
VOID am_onlyone(UINT pid);
```

Ensure that a second copy of an application is not launched. This method simply calls `p_leave(E_FILE_EXIST)`.

It is called during the `am_init` method if the `FLG_APPMAN_ONLYONE` flag was specified and another process of the same name is already running.

The Alarm application is an example of a process which should never have more than one copy running.

★ AM_CHANGE_PRI **Change active object priority**

```
VOID am_change_pri(PR_ACTIVE *pObject, INT priority);
```

Change the priority of the active object with handle `pobject` to the value in `priority`.

The result will be unpredictable if the object is not currently in the application manager's active object queue.

The active object is removed from the application manager's active object queue, the new priority is copied into `active.priority` and the object is then re-inserted with the new priority.

CHAPTER 11

THE ACTIVE CLASS AND ACTIVE OBJECTS

ACTIVE
q priority isactive pcb stat
destroy ao_init ao_cancel ao_abrun ao_queue ao_run

The `ACTIVE` class provides common behaviour for active objects. An active object may be thought of as an event source and is, by definition, any object which has `ACTIVE` as an ancestor in its inheritance tree.

Active objects are fundamental to the operation of event-driven SIBO applications (the overwhelming majority of all SIBO applications). In such an application, virtually all processing is performed within the `ao_run` method of some active object or other.

Although it is not formally an abstract class, the `ACTIVE` class must be subclassed to create a useful active object. `OLIB` and `HWIM` supply a number of subclasses of `ACTIVE` for use by applications. In addition, an application may define one or more application-specific active object classes.

A typical active object corresponds to an asynchronous channel on one of `PLIB`'s I/O devices. An assumption, embodied in `ACTIVE`'s property, is that only one asynchronous event per active object can be outstanding at any one time.

See the *Application Manager* chapter for further information about active objects and event scheduling.

Note that active objects making asynchronous requests on the file server should subclass `FACTIVE` in preference to `ACTIVE`. See the *File Active Objects* chapter for further details.

Precursors

The reader is assumed to understand:

- the `APPMAN` class, in particular the event scheduling mechanisms.
- the `PLIB/EPOC` asynchronous I/O system, waits, signals and semaphores.
- requirements of an event-driven system.
- the `p_enter` and `p_leave` error handling services.

Class definition

The ACTIVE class subclasses ROOT and is defined in the sub-category file *appman.cl* (with generated header file *appman.g*).

```
CLASS active root
Active object superclass for representing event sources
{
  REPLACE destroy      Close I/O channel and free itself
  ADD ao_init          Open I/O channel
  ADD ao_cancel        Cancel outstanding read request
  ADD ao_abrun         Abnormal, p_leave induced, termination of ao_run
  ADD ao_queue         Queue request (normally subclassed)
  ADD ao_run           Provide an opportunity to run

  CONSTANTS
  {
    ! Priorities
    PRIORITY_ACTIVE_POSTER      100
    PRIORITY_ACTIVE_IPCS       80
    PRIORITY_ACTIVE_VOICE      70
    PRIORITY_ACTIVE_WSERV      60
    PRIORITY_ACTIVE_COMMAND    40
    PRIORITY_ACTIVE_SERIAL    20
    PRIORITY_ACTIVE_ALARM      0
    PRIORITY_ACTIVE_FILES     -20
    PRIORITY_ACTIVE_REPEATER   -40
    PRIORITY_ACTIVE_PRINT     -60
    PRIORITY_ACTIVE_COMPUTE   -100
  }

  PROPERTY
  {
    P_QUE q;          queue header
    BYTE priority;    priority compared to other active objects
    UBYTE isactive;   TRUE if there is a request pending
    UBYTE *pcb;       I/O channel
    WORD stat;        I/O completion status
  }
}
```

Property

active.q	Used by the application manager to include an active object in its prioritised queue. It should not be accessed other than by the application manager and an active object's destroy method.
active.priority	Used to determine the object's position in the application manager's prioritised queue. The value, which will normally be one of the priorities listed in the ACTIVE class definition, should be set up prior to sending the application manager an AM_ADD_TASK message.
active.isactive	Should be set to TRUE when the active object is (or will be, on completion of an outstanding asynchronous request - see also active.stat) prepared to receive an AO_RUN message. A common error is to fail to set active.isactive to TRUE when an asynchronous request is made. This will eventually cause <i>stray signal death</i> , described in the <i>Application Manager</i> chapter of this manual. The application manager sets active.isactive to FALSE when it sends the AO_RUN message, avoiding the need for the ao_run method of each individual active object to clear this field.
active.pcb	Normally holds the handle of the device upon which the active object makes its I/O requests. Many of the methods supplied by the ACTIVE class assume that this is a true I/O channel handle.
active.stat	Normally used as the completion status word for an asynchronous request. Only if its value is not E_FILE_PENDING will the application manager's event scheduling loop send an AO_RUN message to this active object (active.isactive must also be TRUE).

ACTIVE methods

J DESTROY

Destroy the instance

VOID destroy(VOID)

Takes the following actions:

- sends itself an AO_CANCEL message to cancel any pending asynchronous request
- removes itself, if necessary, from the application manager's task queue
- closes any I/O channel, whose handle is assumed to be in active.pcb (this is harmless if active.pcb is NULL)
- supersends itself a DESTROY message.

If a subclass uses active.pcb to contain anything other than an I/O channel handle, it should ensure that active.pcb is set to NULL before this method is executed.

AO_INIT

Initialise the instance

VOID ao_init(TEXT *devname, INT mode);

Initialise the active object.

Uses p_open to open a channel to the device specified by devname and mode, writing the channel handle to active.pcb. Calls p_leave if there is an error opening the channel.

The object is not added to the application manager's task queue and no other fields in the property are changed.

J AO_QUEUE

Make a request to run

VOID ao_queue(VOID);

Set active.isactive to TRUE and signal the I/O semaphore by calling p_iosignal without altering the value of active.stat (whose default value is zero). As a result, the object will eventually be sent an AO_RUN message by the application manager.

This method is provided for use by idle object subclasses (see *Idle Objects and the AIDLE Class*). Other active objects would normally subclass this method. All subclasses *must* ensure that the ao_queue method sets active.isactive to TRUE.

J AO_CANCEL

Cancel a request to run

VOID ao_cancel(VOID)

Cancel any outstanding asynchronous request.

Does nothing if active.isactive is not TRUE.

If active.isactive is TRUE then it is re-set to FALSE. If active.pcb is not NULL, it is assumed to be the handle of an I/O channel and a P_FCANCEL request is made to that channel. Regardless of the value of active.pcb, this is followed by a p_waitstat, waiting on active.stat.

The AO_CANCEL message may be received before or after the corresponding request has completed:

- if it is before the completion, the outstanding request is cancelled and the p_waitstat waits for, and absorbs the event which signals the completion of the cancel
- if it is after the completion (but before its processing) the P_FCANCEL request does not generate its own completion event. In this case the p_waitstat consumes the event already generated by the completion of the asynchronous request, effectively discarding it.

Note that the completion result placed in active.stat is likely to be different in the above two cases, but is normally ignored since the status following a cancel is generally not significant.

Since the `destroy` method sends an `AO_CANCEL` message, the `ACTIVE` class contains implicit assumptions that:

- the `ao_cancel` method will never call `p_leave`
- it is safe to send an `AO_CANCEL` message *at any time*, even if there has not been a previous `AO_QUEUE` message.

These assumptions about the cancel service are certainly true at the PLIB level, where a `P_FCANCEL` does not rely on there being an outstanding request (for example `P_FREAD` or `P_FWRITE`). They are also true for all system-supplied subclasses of `ACTIVE`. Subclassers of the `ACTIVE` class should ensure that this assumption remains true.

Active objects that perform operations on files should subclass `FACTIVE` (described in the *File Active Objects* chapter) which provides the correct support for a file system cancel service.

The `ao_cancel` method provided by `ACTIVE` may safely be used by non-I/O subclasses provided they leave `active.stat` at `NULL`.

AO_ABRUN

Handle an error

```
VOID ao_abrun(VOID);
```

Report an error condition arising from a call to `p_leave` in the object's `ao_run` method.

Sends the application manager an `AM_NOTIFYERR` message:

```
p_send5(w_am, 0_AM_NOTIFYERR, w_am->appman.err, w_am->appman.nrid, NULL);
```

and then sets `appman.nrid` to `NULL`.

The application manager has previously set `appman.err` to contain the error number passed as the parameter to `p_leave`, and `appman.nrid` is assumed to be either `NULL`, or an application-specific resource id.

The application manager and its property are accessed via the *magic static* `w_am`, which is assumed to have been initialised (as it is, for example, in the `am_init` method of the `HWIMMAN` subclass of `APPMAN` - see the *HWIM Reference* manual).

The `AO_ABRUN` method may be subclassed to provide more specific error handling. Note that a `CL_CLEAN_LEVEL` message will have been sent to any application manager `CLEANUP` object before the `AO_ABRUN` message is received.

AO_RUN

Process an event

```
INT ao_run(VOID)
```

A default method which simply returns `RUN_ACTIVE_USED`, to signal to the application manager that it has consumed an event. Most active objects will subclass this method.

An active object will only receive an `AO_RUN` message if `active.isactive` is `TRUE` and `active.stat` is not `E_FILE_PENDING`. The application manager sets `active.isactive` to `FALSE` before sending the `AO_RUN` message.

CHAPTER 12

IDLE OBJECTS AND THE AIDLE CLASS

Idle objects

Well-behaved applications should break down long, computationally intensive operations into a sequence of smaller sections processed in *idle time*, so that the application can avoid *going deaf* to window server messages for long periods of time. In other words, these operations should only be allowed to run when no other higher priority work is ready to run (e.g. responding to window server messages).

This is normally achieved by using an active object of low priority (usually `PRIORITY_ACTIVE_COMPUTE`) which will only be run when no other active objects have any work to perform. An active object used for such a purpose is known as an *idle object*.

An idle object will typically leave `active.stat` at a zero value and use the default `ao_queue` method provided by the `ACTIVE` class. Its `ao_run` method is usually subclassed to perform a unit of processing and then (provided processing is not yet complete) send itself an `AO_QUEUE` message.

A partially complete operation may be invalidated by a subsequent event. In such a case the operation should be cancelled and restarted. Such use of an idle object is appropriate, for example:

- in a text processor

word wrapping a line at a time without falling behind in echoing user input in the current line.
- in a spreadsheet

calculating a cell at a time in auto-calculate mode, allowing the user to continue to input during the calculation.

An empty idle object - one whose `ao_run` method does nothing but send itself an `AO_QUEUE` message - will execute its `ao_run` method several hundred times per second.

AIDLE

ACTIVE	AIDLE
q priority isactive pcb stat	
destroy ~~ao_init~~ ao_cancel ao_abrun ao_queue ~~ao_run~~	ao_init ao_run

The `AIDLE` class provides the basic functionality of an idle object.

The supplied `ao_run` method simply requests the application manager to exit from one level of application manager event scheduling by sending the application manager an `AM_STOP` message. In this form it may be used to pause some operation to allow the processing of other events. This usage, which is illustrated in the first example below, can be considered as a means of adding some aspects of idle time processing to code which, for one reason or another, is not suitable for implementation as an active object.

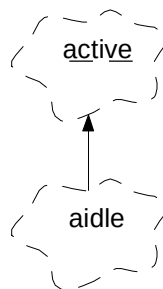
A true idle object must subclass the `ao_run` method to perform the required processing, as described above, and as illustrated in the second example.

Precursors

The reader is assumed to understand:

- the `ACTIVE` class
- the application manager's event scheduling mechanism

Class diagram



Class definition

Defined in sub-category file *appman.cl* (generated header file *appman.g*).

```
CLASS   aidle   active
Idle active object.
{
  REPLACE ao_init      Add itself to active list at low priority
  REPLACE ao_run       Send w_am an O_AM_STOP
}
```

Property

None.

AIDLE methods

J AO_INIT

Initialise

```
VOID ao_init(VOID)
```

Set `active.priority` to `PRIORITY_ACTIVE_COMPUTE` and send the application manager an `AM_ADD_TASK` message.

J AO_RUN

Run

```
INT ao_run(VOID);
```

Send the application manager an `AM_STOP` message and return `RUN_ACTIVE_USED`.

Examples

Pause an operation

Some operations are not suitable for implementation in an active object format. The quicksort algorithm, for example, is recursive and therefore dependent on stacked state information. Since it could take an extended time to sort the data, some action should be taken to ensure that the calling application remains responsive to redraws and user input. A considerable rewrite would, however, be necessary to enable quicksort to execute as a sequence of separate calls to an `ao_run` method.

A more convenient solution in such a case is to insert, at some point which is repeatedly executed, code which pauses execution and allows other events (such as redraws) to be processed. The principle is illustrated in the following code:

```
VOID MakeIdle(VOID)
{
    INT i;
    VOID *aidle;

    ...
    aidle=f_newsend(CAT_DEMO_OLIB,C_AIDLE,O_AO_INIT);
    for (i=0;i<1000;i++)
    {
        ...
        p_send2(aidle,O_AO_QUEUE);
        p_send2(w_am,O_AM_START); /* does not return until AIDLE has run */
        ...
    }
    p_send2(aidle,O_DESTROY);
    ...
}
```

The `AO_QUEUE` message is handled at the `ACTIVE` level, simply setting `active.isactive` to `TRUE` and signalling the I/O semaphore. The send of the `AM_START` message will not return until the `AM_STOP` message is sent by `AIDLE`'s `ao_run` method (which will not run until there are no outstanding events for any active object of priority higher than that of `AIDLE`).

Depending on the nature of the process, it may be more appropriate to pause, say, every tenth, or hundredth, time round the loop. In other words, it is the responsibility of the process to decide when or how often to pause.

Idle time computation

This example illustrates one of the most common forms of idle active object. It is the form that would be used, say, to reformat a portion of text, following the insertion or deletion of characters. It includes the ability to cancel and restart the operation on receipt of a further event (for example, another keypress) which invalidates a partially complete operation.

Note that the `ao_run` method is subclassed so that no use is made of the functionality of `AIDLE`'s `ao_run` method.

```
CLASS      exidle aidle
example idle object
{
    REPLACE ao_cancel      to reset the processing state
    REPLACE ao_run         perform a unit of processing
    PROPERTY
    {
        WORD counter;      records the current processing state
    }
}

METHOD VOID exidle_ao_cancel(PR_EXIDLE *self)
{
    self->exidle.counter=0;
    p_supersend2(self,O_AO_CANCEL);
}
```

```
METHOD INT exidle_ao_run(PR_EXIDLE *self)
{
    p_printf("Processing stage %d",self->exidle.counter++);
    if (self->exidle.counter<3)
        p_send2(self,O_AO_QUEUE);
    return(RUN_ACTIVE USED);
}
```

The active object is run as illustrated below, following the action (say, the insertion of a character) which necessitates the processing.

```
GLREF_D PR_EXIDLE *exidle;

...
p_send2(exidle,O_AO_CANCEL); /* cancel any partially complete processing */
p_send2(exidle,O_AO_QUEUE); /* restart the processing */
...
```

The above code fragment assumes that the active object exists for the lifetime of the application; its creation and destruction would be handled elsewhere.

If the active object is to have a transient existence, it would normally be created (with the appropriate error handling) immediately prior to its use. In this case it would be appropriate for the object to send itself a DESTROY message in its ao_run method on completion of the processing.

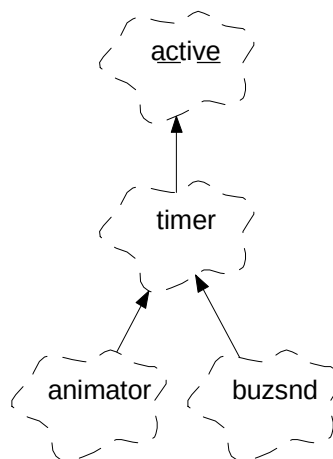
CHAPTER 13

TIMER ACTIVE OBJECT CLASSES

This chapter describes the `TIMER` class and two `TIMER` subclasses, `ANIMATOR` and `BUZSND`.

The `TIMER` class, although not formally an abstract class, must be subclassed to provide a specific `ao_run` method to process the timer expiry.

Class diagram



Precursors

The reader is assumed to understand:

- the `ACTIVE` class.
- the `PLIB/EPOC` timer driver services.
- for `BUZSND`, the `EPOC` sound driver services

TIMER

ACTIVE	TIMER
q priority isactive pcb stat	
destroy ~~ao_init~~ ao_cancel ao_abrun ~~ao_queue~~ ao_run	ao_init ao_queue tm_qabsolute

The `TIMER` class supplies methods to queue both relative and absolute timers. See the *Time, Timers and Dates* chapter of the *PLIB Reference* manual for a description of absolute and relative timers and their differences.

Class definition

Defined in sub-category file *timer.cl* (generated header file *timer.g*).

```
CLASS timer active
The timer active object
{
    REPLACE ao_init      Opens a channel to an asynchronous timer
    REPLACE ao_queue     Queues a relative timer
    ADD tm_qabsolute     Queues an absolute timer
}
```

Property

None.

TIMER methods

AO_INIT

Initialise

```
VOID ao_init(VOID);
```

Open a channel to a timer using the device name "TIM:" by supersending the `AO_INIT` message. This uses the `PLIB p_open` function.

The open timer channel handle will be stored in `active.pcb` if the timer was successfully opened.

Calls `p_leave` on error.

J AO_QUEUE

Queue (relative)

```
VOID ao_queue(UINT lsw, UINT msw);
VOID ao_queue(ULONG time); (conceptual)
```

Queue a request on the timer for a relative timeout (using `active.stat` as the completion status word) where *time* is the required time interval to the timer completion in tenths of a second. The `ULONG time` is actually passed in the message as two `UINT` parameters, `lsw` (least significant word) and `msw` (most significant word).

Sets `active.isactive` to `TRUE`.

Only one timer request (either relative or absolute) may be outstanding at any one time since `active.stat` is used for either timer request.

The timer will receive an `AO_RUN` message on expiry of the timeout.

J TM_QABSOLUTE**Queue (absolute)**

```
VOID tm_qabsolute(UINT lsw, UINT msw);
VOID tm_qabsolute(ULONG time); (conceptual)
```

Queue a request on the timer for an absolute timeout (using `active.stat` as the completion status word) where `time` is the absolute system time at which the timer is to complete. The `ULONG time` is actually passed in the message as two `UINT` parameters, `lsw` (least significant word) and `msw` (most significant word).

Sets `active.isactive` to `TRUE`.

Only one timer request (either relative or absolute) may be outstanding at any one time since `active.stat` is used for either timer request.

The timer will receive an `AO_RUN` message on expiry of the absolute timeout.

ANIMATOR

ACTIVE	TIMER	ANIMATOR
q priority isactive pcb stat		own message interval
destroy ~~ao_init~~ ao_cancel ao_abrun ~~ao_queue~~ ~~ao_run~~	tm_qabsolute ~~ao_init~~ ao_queue	ao_init ao_run

The `ANIMATOR` class supplies the functionality to send a message to an object at regular intervals, the message, object and time interval being specified at initialisation.

Since the intention is that `ANIMATOR` will be used to drive an animation sequence, it runs at a priority which is much higher than that normally used by timers but which, at the same time, is less than `PRIORITY_ACTIVE_WSERV` so that window server events - particularly redraw events resulting from the animation - are not blocked. The actual priority is set to `PRIORITY_ACTIVE_WSERV - 1`.

Class definition

Defined in sub-category file `timer.cl` (generated header file `timer.g`).

```
CLASS animator timer
Sends regular messages eg to drive animation
{
  REPLACE ao_init
  REPLACE ao_run
  TYPES
  {
    typedef struct          Must be in property order
    {
      PR_ROOT *own;         send messages to this object
      INT message;          message number to send
      INT interval;         delay between subsequent messages in tenths of a second
      INT first;            delay before first message back in tenths of a second
    } IN_ANIMATOR;
  }
  PROPERTY
  {
    PR_ROOT *own;          send messages to this object
    INT message;           the number of the message to send
    INT interval;          interval between messages
  }
}
```

Property

<code>animator.own</code>	the handle of an object to which messages will be sent (this will normally be the object which owns the instance of <code>ANIMATOR</code>)
<code>animator.message</code>	the number of the message to be sent to <code>animator.own</code>
<code>animator.interval</code>	the time interval, in tenths of a second, between the sending of two successive messages

ANIMATOR methods**AO_INIT****Initialise**

```
VOID ao_init(IN_ANIMATOR *pin);
```

Initialise the animator object by:

- supersending an `AO_INIT` message, which opens a timer channel
- setting `active.priority` to `PRIORITY_ACTIVE_WSERV-1` and sending the application manager an `AM_ADD_TASK` message to add the animator object to the application manager's active object task queue.
- setting up `animator.own`, `animator.message` and `animator.interval` from the data pointed to by `pin`.
- sending itself an `AO_QUEUE` message with the interval timeout value specified by `pin->first`. In effect, this defines the time interval before the animator object receives the first `AO_RUN` message.

AO_RUN**Run**

```
INT ao_run(VOID)
```

Handle the completion of the timer request by:

- sending the message `animator.message` to the object `animator.own`
- sending itself an `AO_QUEUE` message with the timeout value specified by `animator.interval`
- returning `RUN_ACTIVE_USED`

BUZSND

ACTIVE	TIMER	BUZSND
<code>q</code> <code>priority</code> <code>isactive</code> <code>pcb</code> <code>stat</code>		<code>snd</code> <code>sndrep</code> <code>sndnum</code> <code>snddelay</code> <code>sndvolume</code> <code>h_done</code> <code>m_done</code>
<code>destroy</code> <code>ao_init</code> <code>ao_cancel</code> <code>ao_abrun</code> <code>ao_queue</code> <code>ao_run</code>	<code>tm_qabsolute</code> <code>ao_init</code> <code>ao_queue</code>	<code>ao_init</code> <code>ao_cancel</code> <code>ao_queue</code> <code>ao_run</code>

The `BUZSND` class provides the means for an application to generate alarm sound sequences.

Each alarm sequence consists of one of two possible sounds repeated eight times with a two second interval between each sound. The volume of the sound is increased with each repetition.

The sound itself can be either a 'rings' sequence or a 'chimes' sequence and is selected when an AO_INIT message is received.

The sounds are generated by means of the sound driver as described in the *Sound* chapter of the *I/O Devices Reference* manual. Since only one user may have access to the sound system at any one time, BUZSND serialises multiple access requests from different applications. To avoid monopolising the sound driver, the channel is opened and closed for each sound in the sequence.

This active object is interesting, in that it may have an outstanding request on either the sound or the timer channel, but not both at the same time; the two channels alternately use `active.isactive` and `active.stat`. The descriptions of the `ao_cancel` and the `ao_run` methods include sample code to clarify the explanation of the techniques involved.

Class definition

Defined in sub-category file *timer.cl* (generated header file *timer.g*).

```

CLASS buzsnd timer
  Buzzer sound generator
  {
    REPLACE ao_init          Init timer and add to appman task list
    REPLACE ao_cancel        Cancel the timer or sound
    REPLACE ao_queue         Start a sound
    REPLACE ao_run           Handle next step of sound sequence
    PROPERTY
      {
        UBYTE *snd;          Open sound channel handle
        UWORD sndrep;        Number of repeats to do
        UWORD sndnum;        Which sound number to use
        UWORD snddelay;      Delay between repeats
        UWORD sndvolume;     For SND: growing volume
        PR_ROOT *h_done;     Handle to receive completion message
        UWORD m_done;        Method number for above
      }
  }

```

Property

<code>buzsnd.snd</code>	the channel handle of the sound driver, while the sound driver is being used.
<code>buzsnd.sndrep</code>	the remaining number of repetitions in the current sound sequence
<code>buzsnd.sndnum</code>	which sound to use (0 for a 'rings' sequence or 1 for a 'chimes' sequence)
<code>buzsnd.snddelay</code>	the time, in tenths of a second, of the delay between successive sounds
<code>buzsnd.sndvolume</code>	the volume of the current sound in the sequence
<code>buzsnd.h_done</code>	NULL, or the handle of the object to which a message is sent on completion of the sound sequence
<code>buzsnd.m_done</code>	the message number to be sent on completion of the sound sequence

BUZSND methods

AO_INIT

Initialise

```
VOID ao_init(UINT sndnum);
```

Open a timer device channel by supersending an AO_INIT message to the TIMER superclass, store `sndnum` (either a 0 for a 'rings' sequence or a 1 for a 'chimes' sequence) in `buzsnd.sndnum`, set `appman.priority` to `PRIORITY_ACTIVE_REPEATER` and send the application manager an `AM_ADD_TASK` message.

Calls `p_leave` on error, typically with the error `E_GEN_NOMEMORY`.

J AO_CANCEL**Cancel**

```
VOID ao_cancel(VOID)
```

Cancel either the timer or the sound driver, whichever is currently running.

If the sound channel is open (`buzsnd.snd` is non-zero), any outstanding request is cancelled using the PLIB I/O `P_FCANCEL` service. The sound channel is closed and `buzsnd.snd` is set to `NULL` so that further closes are harmless.

In all cases the `ao_cancel` method then supersends an `AO_CANCEL` message to cancel any outstanding timer request. As with all `ao_cancel` methods, this is harmless if there is no outstanding request.

Finally, `buzsnd.sndrep` and `buzsnd.snddelay` are set to their starting values of 8 and 20 (i.e. 20 by 1/10th second) respectively, and `buzsnd.sndvolume` is set to one of two starting values depending on whether the user has set the machine to generate loud or quiet sounds.

```
METHOD VOID buzsnd_ao_cancel(PR_BUZSND *self)
{
    if (self->buzsnd.snd) /* destroy may call cancel if init fails */
    {
        p_iow2(self->buzsnd.snd,P_FCANCEL); /* all harmless if not running */
        p_waitstat(&self->active.stat); /* see note ++ */
        self->active.isactive=FALSE; /* see note ++ */
        p_close(self->buzsnd.snd);
        self->buzsnd.snd=NULL;
    }
    p_supersend2(self,O_AO_CANCEL); /* timer cancel is harmless if not running */
    self->buzsnd.sndrep=8;
    self->buzsnd.snddelay=20; /* in tenths of a second */
    self->buzsnd.sndvolume=(p_getsnd()&E_SOUND_LOUD)?
        (E_SOUND_MIN_VOLUME-1)*2+1:(E_SOUND_MIN_VOLUME)*2+1;
}
```

While the calculation for `buzsnd.sndvolume` in the last line of the code is obscure, it does represent the most efficient way (in conjunction with the `ao_run` method) of calculating a gradually increasing volume.

Note

For the byte-conscious programmer, these two lines (marked `++`) are not strictly necessary. These actions will be performed within the `p_supersend` of an `AO_CANCEL` which follows a few lines further down. This relies on the fact that the timer and sound channels share the same status word and never have simultaneous outstanding requests.

J AO_QUEUE**Queue**

```
VOID ao_queue(PR_ROOT *handle, UINT method);
```

Start the generation of a sound sequence.

Any currently outstanding sound being generated is cancelled by sending itself an `AO_CANCEL` message, which also resets the sound control parameters `buzsnd.sndvolume`, `buzsnd.sndrep` and `buzsnd.snddelay` to their starting values. The `handle` and `method` values are stored in `buzsnd.h_done` and `buzsnd.m_done` respectively.

The sound generation sequence is started off by setting `active.isactive` to `TRUE` and calling `p_iosignal`.

The `ao_run` method will be called by the active object scheduling code in the application manager when no other events of higher priority are outstanding.

AO_RUN**Run**

```
INT ao_run (VOID)
```

Make alternate requests for a sound or a timeout until the sound sequence is complete.

If the last to run was the sound driver:

- the sound driver is closed and `buzsnd.snd` is set to `NULL`
- if the sound sequence is not complete, the timer is queued by supersending an `AO_QUEUE` message with a timeout as defined by `buzsnd.snddelay` (2 seconds).

- if the sound sequence is complete and `buzsnd.h_done` is non-zero, a `buzsnd.m_done` message is sent to `buzsnd.h_done`.

If the last to run was the timer:

- an attempt is made to open the sound driver
- if the sound driver is currently busy, a five second timeout is queued
- if the sound driver has been disabled then the sound sequence is deemed to have completed and the completion message is sent, as described above
- if the sound driver cannot be opened for any other reason (e.g. insufficient memory being available) then `p_leave` is called.
- if the sound driver is opened successfully, the volume is adjusted so that it gradually becomes louder and an asynchronous request is made to generate an alarm sound (this branch requires `active.isactive` to be explicitly set to `TRUE`)

In all cases the method returns `RUN_ACTIVE_USED`.

```
METHOD buzsnd_ao_run(PR_BUZSND *self)
{
    INT ret;
    UWORD delay;
    UBYTE bb[10];
    E_SOUND c;

    if (!self->buzsnd.snd)
    { /* last to run was the timer */
        bb[0]='S';bb[1]='N';bb[2]='D';bb[3]=':';bb[4]=0;
        ret=p_open(&self->buzsnd.snd,&bb[0]);
        if ((ret==E_FILE_LOCKED) || (ret==E_GEN_INUSE))
        { /* busy - try again later */
            delay=50; /* 5 seconds */
            p_supersend4(self,0_AO_QUEUE,delay,0); /* last 2 parameters are a LONG */
        }
        else
        {
            if (ret==E_GEN_FAIL) /* sound driver disabled */
                goto sendOwnerDone; /* immediate completion (silent alarm) */
            f_leave(ret);
            p_iow3(self->buzsnd.snd,P_FSENSE,&c);
            c.volume=(self->buzsnd.sndvolume--)>>1;
            p_iow3(self->buzsnd.snd,P_FSET,&c);
            p_ioc4(self->buzsnd.snd,E_FALARM,&self->active.stat,&self->buzsnd.sndnum);
            self->active.isactive=TRUE;
        }
    }
    else
    { /* last to run was the sound */
        p_close(self->buzsnd.snd);
        self->buzsnd.snd=0;
        if (--self->buzsnd.sndrep)
        {
            p_supersend4(self,0_AO_QUEUE,self->buzsnd.snddelay,0);
            /* last 2 pars are a LONG */
        }
        else
        {
            if (self->buzsnd.h_done)+
                p_send2(self->buzsnd.h_done,self->buzsnd.m_done);
        }
    }
    return(RUN_ACTIVE_USED);
}
```


CHAPTER 14

FILE ACTIVE OBJECTS

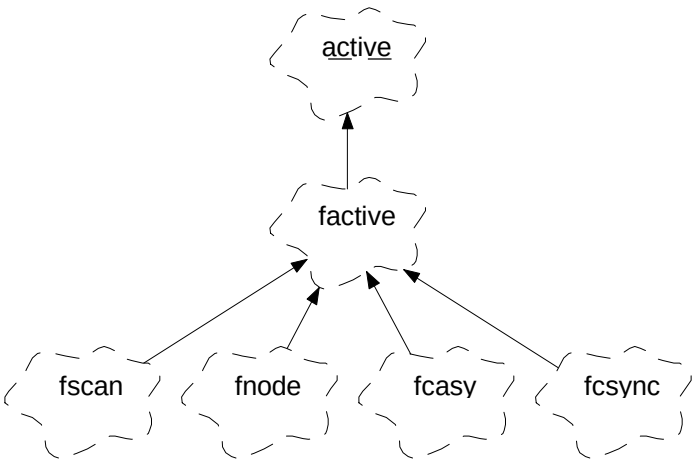
This chapter describes the `FACTIVE` class (subclassed by all active objects which perform asynchronous operations on files) and its `FSCAN`, `FNODE`, `FCASY` and `FCSYNC` subclasses.

Precursors

The reader is assumed to understand:

- the `ACTIVE` class
- the file server and file system services as described in the *Files* chapter of the *PLIB Reference manual*

Class diagram



FACTIVE

ACTIVE	FACTIVE
q priority isactive pcb stat	owner
destroy ao_init ao_cancel ao_abrun ao_queue ao_run	fa_close ao_init ao_cancel ao_abrun

The `FACTIVE` class provides the basic functionality of all active objects that perform operations on files.

Although not formally an abstract class, `FACTIVE` must be subclassed to be useful. The subclass will, in general, need to supply at least an `ao_queue` and an `ao_run` method.

Although the file server does not support a cancel service (see the *Files* chapter of the *PLIB Reference* manual) `FACTIVE` supplies an `ao_cancel` method which simulates the cancelling of an outstanding asynchronous request. This allows the coding of file-related active objects to follow the style of coding for other active objects which, for example, routinely send an `AO_CANCEL` message prior to destruction of the instance.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```
CLASS    factive active
File active object superclass
{
  REPLACE ao_init           Set priority and am_add_task
  REPLACE ao_cancel        Simulate cancel and close file
  REPLACE ao_abrun         Cancel and supersend
  ADD fa_close             Close and set pcb to NULL
  PROPERTY
  {
    UBYTE *owner;          Owinging object
  }
}
```

Property

<code>factive.owner</code>	the handle of the owinging object, for use by subclasses, for example, to report the completion of file activity - not used by <code>FACTIVE</code>
----------------------------	---

FACTIVE methods

★ AO_INIT

Initialise

```
VOID ao_init(UBYTE *owner);
```

Initialise the file active object by:

- setting `factive.owner` to `owner`, the handle of the owinging object
- setting `active.priority` to `PRIORITY_ACTIVE_FILES`
- sending the application manager an `AM_ADD_TASK` message to add the file active object to the application manager's active object task queue

★ AO_CANCEL

Cancel

```
VOID ao_cancel(VOID);
```

Cancel any outstanding file server event and close any open file. This is harmless if there is no outstanding event.

Note that the cancellation is simulated since the file server does not support a cancel service (see *Asynchronous file operations* in the *Files* chapter of the *PLIB Reference* manual). The end effect is, however, indistinguishable from a true cancel in that, if a file server event is outstanding, `active.stat` is set to `E_FILE_CANCEL` and `active.isactive` is set to `FALSE`.

In addition, this method sends an `FA_CLOSE` message to ensure that any open file is closed.

AO_ABRUN**Handle error**

```
VOID ao_abrun(VOID);
```

Send itself an AO_CANCEL message to cancel any outstanding file activity and close any open file and then supersend an AO_ABRUN message.

This may leave with any error that could arise in the superclass ao_abrun method.

★ FA_CLOSE**Close any open file**

```
VOID fa_close(VOID);
```

Close (with p_close) any open file whose handle is in active.pcb, and set active.pcb to NULL.

This method does not call p_leave. It is a requirement that any subclass must not call p_leave.

FSCAN

ACTIVE	FACTIVE	FSCAN
q priority inactive pcb stat	owner	flags index pname match delim finfo crk pcbarr name
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	~~fa_close~~ ao_init ao_cancel ao_abrun	fs_matchname fs_fscan ao_queue ao_run fa_close fs_fscan_end fs_dirname fs_filename fs_end_dirlist

FSCAN is an abstract subclass of FACTIVE which provides the basic mechanisms for scanning a filing system by reading the content of one or more directory files. It is designed to be independent of any particular filing system and can be used, for example, with either Macintosh or DOS-compatible filing systems.

Subclasses of FSCAN may be used to scan the filing system to select files which match any of a variety of criteria. The subclass must supply the action(s) required when a matching directory or file is found.

It may be noted that on completion of the scan, the whole of each relevant directory file will always have been read; files are eliminated by the matching process within the FSCAN code. This allows, for example, subclasses of FSCAN to extract multiple file specifications in one scan of the directory file or to build file name extension lists.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```
CLASS fscan factive
Scans the directory structure for files/dirs and creates variable str arrays
{
  REPLACE ao_queue           Queue a directory read
  REPLACE ao_run             Process read completion
  REPLACE fa_close           Close all open directory files
  ADD fs_matchname           Match a found name
  ADD fs_fscan              Start a directory scan
  DEFER fs_fscan_end         No more files/dirs
  DEFER fs_dirname           Next directory name from scan
  DEFER fs_filename         Next file name from scan
  DEFER fs_end_dirlist       End of that subdir

  CONSTANTS
  {
    ! Which types of files caller wants
    FS_WRITABLE      P_FAWRITE  opposite of DOS Read-only attribute
    FS_HIDDEN        P_FAHIDDEN as DOS Hidden attribute
    FS_SYSTEM        P_FASYSTEM as DOS System attribute
    FS_DIRECTORIES   P_FADIR
    FS_MODIFIED      P_FAMOD    as DOS Archive attribute
    FS_ALL_FILES      P_FAREAD
    FS_FILE_TYPE      (FS_HIDDEN|FS_SYSTEM)
    FS_INCLUDE_SUBDIRECTORIES 0x1000 Want files from subdirectories
    FS_INCLUDE        0x2000      Called dirname because of include
    FS_END_DIRLIST    0x4000      Called end dir list
    FS_PARSE_NAME     0x8000      Set if generated name to be parsed
    FS_MAX_DIRLEVELS  32          Max number of sub dir levels
  }

  PROPERTY
  {
    UWORD flags;           Controlling flags
    UWORD index;           Subdir array index
    UBYTE *pname;          Pointer into name[] for read
    UBYTE *match;          Pointer to file name match
    UBYTE delim[2];
    P_INFO finfo;          File Info
    P_FPARSE crk;          Parsed file info
    UBYTE *pcbarr[FS_MAX_DIRLEVELS];
    UBYTE name[P_FNAME_SIZE];
  }
}
```

Property

<code>fscan.flags</code>	controlling flags, some of which should be set up by the owner before sending an <code>FS_FSCAN</code> message
<code>fscan.index</code>	the index of the first free entry in the <code>fscan.pcbarr</code> array. It should not be accessed by any subclass.
<code>fscan.pname</code>	a pointer to the file name and extension within the full file specification in the <code>fscan.name</code> buffer. Owners and subclasses should treat this as a read-only field.
<code>fscan.match</code>	a pointer to a string used to match file names during a scan. It may be set up by an owner before sending an <code>FS_FSCAN</code> message.
<code>fscan.delim</code>	temporary storage for the directory delimiter character (assumed to be a single character). It should not be accessed by any subclass.
<code>fscan.finfo</code>	the PLIB <code>P_INFO</code> data for the current file, that is, the file whose name has last been read from a directory file. This information may be read, but should not be modified, by an owner.

<code>fscan.crk</code>	provided the <code>FS_PARSE_NAME</code> flag is set in <code>fscan.flags</code> , this contains the PLIB <code>P_PARSE</code> data for the current file, whose full file specification is in the <code>fscan.name</code> buffer. This information may be read by an owner.
<code>fscan.pcbarr</code>	an array of up to <code>FS_MAX_DIRLEVELS</code> handles of open directory files. This should not be accessed by a subclass.
<code>fscan.name</code>	the full file specification of the current file. An owner may read this name directly or may read only the file name via <code>fscan.pname</code> .

FSCAN methods

AO_QUEUE

Directory read

`VOID ao_queue(VOID);`

Queue a read on the current directory file, setting `active.isactive` to `TRUE`.

The action may be modified by the `ao_run` method resulting from a previous read:

- If the previous read produced the name of a directory file and the scan is to extend into nested subdirectories (`fscan.flags` includes `FS_INCLUDE_SUBDIRECTORIES`) the current value of `active.pcb` is stored in the `fscan.pcbarr` array and the new subdirectory is opened before the read request is made. A maximum of 32 levels of subdirectory may be open at any one time.
- If the previous read detected that there were no more files in the current directory and directories have been nested, then the most recently nested subdirectory handle is restored into `active.pcb` from the `fscan.pcbarr` array before the read request is made.

Any error causes `p_leave` to be called.

AO_RUN

Process read completion

`INT ao_run(VOID);`

Process the completion of the read of a file name from a directory file and return `RUN_ACTIVE_USED`.

If the read completed with an `E_FILE_EOF` error, indicating that there are no further entries in the current directory file:-

- the file is closed and `active.pcb` is set to `NULL`.
- The delimiter character is picked up (the last character before the filename) and placed in `fscan.delim`.

If directories are nested:-

- an `FS_END_DIRLIST` message is sent to indicate the end of a directory, but not the end of the scan.
- the run method completes and returns `RUN_ACTIVE_USED`

If directories are *not* nested:-

- an `FS_FSCAN_END` message is sent to indicate the end of the scan.
- the run method completes and returns `RUN_ACTIVE_USED`

If the read completed successfully:-

- if the file name is a volume name, the name is discarded and a new name is requested by calling the FSCAN `ao_queue` method directly.
- if *none* of the flags `FS_WRITABLE`, `FS_HIDDEN`, `FS_SYSTEM`, `FS_MODIFIED`, `FS_ALL_FILES` is set, the name is discarded and a new name is requested by calling the FSCAN `ao_queue` method directly.
- if the file is a (DOS) `.` or `..` directory file, the name is discarded and a new name is requested by calling the FSCAN `ao_queue` method directly.

- if the file is a directory file and either of the `FS_DIRECTORIES` or `FS_INCLUDE_SUBDIRECTORIES` flags is set (meaning that there is interest in the directory file name itself or in subdirectories) then an `FS_DIRNAME` message is sent. If neither flag `FS_DIRECTORIES` or `FS_INCLUDE_SUBDIRECTORIES` is set, the name is discarded and a new name is requested by calling the `FSCAN ao_queue` method directly

if the file is *not* a directory file:-

- flag is set, then the file name in `fscan.name` is parsed (using `p_fparse`) with the parsed file name information written to `fscan.crk`.
- an `FS_MATCHNAME` message is sent. If this returns `FALSE`, indicating that the file name does not match the specified attributes and (wildcard) name, the name is discarded and a new name is requested by calling the `FSCAN ao_queue` method directly. If the `FS_MATCHNAME` message returns `TRUE` then the read of a valid, matching, file name is indicated by sending an `FS_FILENAME` message.
- the run method completes and returns `RUN_ACTIVE_USED`.

All errors, other than the `E_FILE_EOF` error discussed above, result in `p_leave` being called.

★ FA_CLOSE

Close directory files

```
VOID fa_close(VOID);
```

Supersend an `FA_CLOSE` message and close all open directory files.

★ FS_MATCHNAME

Match a found name

```
INT fs_matchname(VOID);
```

Check that the file matches the specified combination of `FS_MODIFIED`, `FS_HIDDEN` and `FS_SYSTEM` flags. If these tests succeed the file name (pointed to by `fscan.pname`) is tested for a match with the (wildcard) name pointed to by `fscan.match`.

Note that the name match is case-sensitive. Since `FSCAN` is designed to work independently of any particular filing system, matching uses a true wildcard string match (as opposed to a DOS-specific file system wildcard match) to select files. Thus the wildcard string `"**"` will select all files (including those with an extension, unlike DOS). This also means that more than one part of a file name can be wildcarded with the `'*'` character, to find, for example, files matching `"*fred.*"`.

Returns `TRUE` if there is a match, otherwise `FALSE`.

This method is called from within the `ao_run` method.

A subclass may replace this method to provide an alternative name matching algorithm.

FS_FSCAN

Start a directory scan

```
VOID fs_fscan(UBYTE *path);
```

Start the scan of the file system for directories and files.

The file specification pointed to by `path` is parsed to extract the directory in which the scan is to start. Provided this initial path name does not need to be preserved, `path` may point to a file specification in `fscan.name` (this field is overwritten during the scan).

If the specified directory is opened successfully, an `AO_QUEUE` message is sent to start the scan.

Any error causes `p_leave` to be called.

It is assumed that `fscan.match` and `fscan.flags` have previously been set up either by a subclass or by the owner as is done, for example, by many of the methods (such as `fman_rename`) of the `FMAN` class, described in the *File Management Classes* chapter.

The `fscan.match` field should be set to point to a suitable wildcard string.

The `fscan.flags` field should contain a bitwise combination of one or more of the following values:

<code>FS_ALL_FILES</code>	include all non-directory files that are neither hidden nor system files
<code>FS_WRITABLE</code>	as <code>FS_ALL_FILES</code> , but excludes write-protected files (see below)
<code>FS_HIDDEN</code>	include hidden files
<code>FS_SYSTEM</code>	include system files
<code>FS_DIRECTORIES</code>	include directory files
<code>FS_MODIFIED</code>	exclude unmodified files (include only files that have <code>P_FAMOD</code> set)
<code>FS_INCLUDE_SUBDIRECTORIES</code>	extend the scan to subdirectories
<code>FS_PARSE_NAME</code>	parse file names before reporting them

The supplied `fs_matchname` method does not test the `FS_WRITABLE` flag which therefore has the same effect as the `FS_ALL_FILES` flag. A subclass `fs_matchname` method may implement this flag as follows:

```
if ((self->fscan.flags&FS_WRITABLE) && !(self->fscan.finfo.status&P_FAWRITE))
    return(FALSE);
return(p_supersend2(self, O_FS_MATCHNAME));
```

Deferred FSCAN methods

FS_FSCAN_END

Scan completion

```
VOID fs_fscan_end(VOID);
```

A deferred method indicating the end of the scan and that there are no more files or subdirectories to scan into or report back.

This message is sent from within the `ao_run` method. By the time it is sent, all levels of directory files will have been closed.

FS_DIRNAME

Next directory name

```
VOID fs_dirname(VOID);
```

A deferred method indicating that a directory file matching the initial specification has been found.

This message is sent from within the `ao_run` method, provided that `fscan.flags` includes either `FS_DIRNAME` or `FS_INCLUDE_SUBDIRECTORIES`.

On receipt of this message `fscan.finfo` contains the file information for the directory file, `fscan.name` contains its full file specification and the `fscan.pname` points to the directory file name within the full file specification. These fields should be regarded as read only.

In order to obtain further files or directories, this method must restart the scan by sending an `AO_QUEUE` message.

FS_FILENAME

Next file name

```
VOID fs_filename(VOID);
```

A deferred method indicating that a file name matching the initial specification has been found. This message is sent from within the `ao_run` method.

On receipt of this message `fscan.finfo` contains the file information for the file, `fscan.name` contains its full file specification and the `fscan.pname` points to the file name within the full file specification. If `fscan.flags` includes `FS_PARSE_NAME`, then `fscan.crk` contains the parsed file name information. These fields should be regarded as read only.

In order to obtain further files or directories, this method must restart the scan by sending an `AO_QUEUE` message.

FS_END_DIRLIST**End of subdirectory**

```
VOID fs_end_dirlist(VOID);
```

A deferred method indicating that the end of a subdirectory has been reached. This message is sent from within the `ao_run` method provided that `fscan.flags` includes `FS_INCLUDE_SUBDIRECTORIES`. It is only reported at the end of a nested subdirectory and not when the end of the directory in which the scan started is reached. The directory file will be closed before this message is sent.

On receipt of this message `fscan.name` contains the full file specification of the directory file and the `fscan.pname` points to the file name within the full file specification. The information in `fscan.finfo` and `fscan.crk` is not valid.

In order to obtain further files or directories the scan must be restarted by sending an `AO_QUEUE` message at some point.

FNODE

ACTIVE	FACTIVE	FNODE
q priority isactive pcb stat	owner	flags pname oldname pcb
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	~~fa_close~~ ao_init ao_cancel ao_abrun	fn_list ao_queue ao_run fa_close fn_end_list fn_nodename

In EPOC there are multiple filing systems, three of which are *ROM::*, *LOC::* and *REM::*. The `FNODE` abstract class provides the basic mechanisms to generate filing system node and device lists.

The `FNODE` methods read an item in the filing system (node) list and, if that node supports multiple devices (drives) read the device list to generate a *node::device:* name. Filing systems (such as *ROM::*) that do not support multiple devices are ignored.

It is possible to restrict the list to the *LOC::* filing system.

Since filing systems are dynamic under EPOC, the names generated may vary between invocations.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```

CLASS fnode factive
Node/Device name list generator
{
  REPLACE ao_queue           Read from appropriate channel
  REPLACE ao_run             Process read completion
  REPLACE fa_close           Closes both open channels
  ADD fn_list                Start to generate a list
  DEFER fn_end_list          Handle list generate completion
  DEFER fn_nodename          Process an item in the list
  CONSTANTS
  {
    FNODE_NODE_ARRAY 0x01      Reading node list
    FNODE_DEVICE_ARRAY 0x02    Reading device list
    FNODE_LOCAL_ONLY 0x04      Read local node list only
  }
  PROPERTY
  {
    UWORD flags;              Controlling flags
    UBYTE *pname;             Where to read into
    UBYTE *oldname;           Node read ptr
    UBYTE *pcb;               Node pcb while reading device list
  }
}

```

Property

<code>fnode.flags</code>	internal controlling flags
<code>fnode.pname</code>	the offset into the data buffer of where to write the node or device name. It should not be accessed by an owning object.
<code>fnode.oldname</code>	the offset into the data buffer of where to read the next node name. It should not be accessed by an owning object.
<code>fnode.pcb</code>	while reading a device list, the handle of the opened node list file is saved here. It should not be accessed by an owning object.

FNODE methods

★ AO_QUEUE

Read from appropriate channel

```
VOID ao_queue(VOID);
```

Queue a read on either the node list or device list, depending on which is appropriate at the time and set `active.isactive` to TRUE.

Note that node information as would be written to a `P_NINFO` structure is not requested in the read.

AO_RUN

Process read completion

```
INT ao_run(VOID);
```

Process the completion of a read that was initiated by the `ao_queue` method and return `RUN_ACTIVE_USED`. The read may have been from either the node list or a device list. The two cases are discussed under their respective headings.

Node List Read

The completion status is tested for an `E_FILE_EOF` error, indicating that no further nodes exist; if this is the case, the scan is terminated by closing the node list and sending an `FN_END_LIST` message. Otherwise, the node is checked for validity:-

- the node must support multiple devices
- if the `FNODE_LOCAL_ONLY` flag is present, only the *LOC*:: node is valid.

For a valid node, the device list for that node is opened and an `AO_QUEUE` message sent to read an item from the device list, otherwise an `AO_QUEUE` message is sent to read a further item from the node list.

Device List Read

The completion status is tested for an `E_FILE_EOF` error, indicating that no further devices exist on the current node. If the completion status is `E_FILE_EOF`:-

- the device list is closed
- an `AO_QUEUE` message is sent to read the next item from the *node* list.

If the read completed successfully, an `FN_NODENAME` message is sent.

The owner is responsible for re-queuing a read on the node/device list by sending an `AO_QUEUE` message at some future point after sending the `FN_NODENAME` message.

For all errors other than the `E_FILE_EOF` errors discussed above, `p_leave` is called.

★ FA_CLOSE

Close both open channels

```
VOID fa_close(VOID);
```

Close any open node and device lists setting both the open handles to `NULL`. The node list is closed by supersending an `FA_CLOSE` message; the device list is closed by calling `p_close`.

FN_LIST

Start list generation

```
VOID fn_list(UBYTE *pname, UINT flags);
```

Start the scan to generate *node::device:* names by opening the node list and sending an `AO_QUEUE` message.

The user-supplied buffer at `pname` is assumed to be at least `P_FNAME_SIZE` bytes in length. It is used as the output buffer for each generated name and hence, *must* be preserved until an `FN_END_LIST` message is received. The user may access this buffer only when no read is outstanding, for example, during the processing of the deferred `FN_NODENAME` message.

The value of `flags` may be either `FNODE_LOCAL_ONLY` to read only the *LOC::* filing system, or `NULL` to read all filing system device lists.

Deferred FNODE methods

FN_END_LIST

Handle completed list

```
VOID fn_end_list(VOID);
```

A deferred method indicating that the scan is complete and that all *node::device:* names have been generated.

On receipt of this message the node and device lists will have been closed. The user-supplied buffer (see the `fn_list` method) may now be discarded.

FN_NODENAME

Process a list item

```
VOID fn_nodename(VOID);
```

A deferred method indicating that a *node::device:* name has been generated. The name may be read from the user-supplied buffer (see the `fn_list` method).

The user is responsible for restarting the scan for the next name by sending an `AO_QUEUE` message.

FCASY

ACTIVE	FACTIVE	FCASY
q priority isactive pcb stat	owner	recvname flags
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	~~fa_close~~ ao_init ~~ao_cancel~~ ao_abrun	ao_cancel ao_run ao_queue fa_close fc_write fc_open fc_request_comp

The FCASY abstract class subclasses FACTIVE to provide a set of methods to perform a segmented read or write of a single file. In other words, it allows a file to be read or written to in a finite number of discrete portions. This permits a large file to be processed (which would otherwise be impossible due to memory constraints)

FCASY does not support a combination of reading and writing to the same file.

Typical uses are in copying a file or in XMODEM file transfer, where the source and target files are each represented by a separate instance of a subclass of FCASY.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```

CLASS fcsy factive
File copy/save/load asynchronous routines
{
  REPLACE ao_cancel      Abandon and clean up correctly
  REPLACE ao_run         Maybe cleanup and send itself fc_request_comp
  REPLACE ao_queue       Async file read
  REPLACE fa_close       Close file, set recname to NULL
  ADD fc_write           Async file write
  ADD fc_open            Open/create file for load/save
  DEFER fc_request_comp  That async request has completed

  CONSTANTS
  {
    ! Display info flags
    FCOPY_DISP_SFTYPE 0x01
    FCOPY_DISP_SFSIZE 0x02
    FCOPY_DISP_SFDATE 0x04
    FCOPY_DISP_SFNAME 0x08
    FCOPY_DISP_RFTYPE 0x10
    FCOPY_DISP_RFSIZE 0x20
    FCOPY_DISP_RFDATE 0x40
    FCOPY_DISP_RFNAME 0x80
    FCOPY_DISP_BLKSIK 0x100
    FCOPY_DISP_BLKNO 0x200
    FCOPY_DISP_PROTOCOL 0x400
    ! Controlling Flags
    FCASY_READ_QUEUED 0x01
    FCASY_WRITE_QUEUED 0x02
  }
}

```

```
TYPES
{
    typedef struct
    {
        UWWORD flags;           FCOPY_DISP_... flags indicate which fields are
valid        UWWORD protocol;   which protocol being used
        UWWORD blksize;         Size of each block
        UWWORD blkno;          Current block number
        UWWORD sftype;         Source file type
        UWWORD rftype;         Receive file type
        ULONG sfsz;            Source file size
        ULONG rfsz;            Receive file size
        ULONG sfdate;          Source file last modification date
        ULONG rfdate;          Receive file last modification date
        UBYTE *sfname;         Source file name (source for reads)
        UBYTE *rfname;         Receive file name (destination for writes)
    } FCOPY_DISP;
}

PROPERTY
{
    UBYTE *recvname;           Receive file name
    UWWORD flags;              Controlling flags
}
}
```

Property

<code>fcasy.recvname</code>	a pointer to the name of the file being written to (NULL if the file is being read)
<code>fcasy.flags</code>	internal controlling flags; while a request is outstanding, contains a value indicating the nature of the request (either <code>FCASY_READ_QUEUED</code> or <code>FCASY_WRITE_QUEUED</code>)

FCASY methods

★ AO_CANCEL

Cancel request

```
VOID ao_cancel(VOID);
```

Cancel any outstanding read or write by supersending an `AO_CANCEL` message.

If the instance represents a file that is opened for writing, the file is then explicitly deleted.

★ AO_QUEUE

Read request

```
VOID ao_queue(UBYTE *buf, UWWORD *plen);
```

Queue a read of `*plen` bytes into `buf` from the file whose channel handle is in `active.pcb`.

Sets `active.isactive` to TRUE and sets the `FCASY_READ_QUEUED` flag in `fcasy.flags` to indicate a read request.

Since the read is asynchronous, the memory pointed to by `buf` and `plen` must be preserved until the request completes.

★ FC_WRITE

Write request

```
VOID fc_write(UBYTE *buf, UWWORD *plen);
```

Queue a write of `*plen` bytes from `buf` to the file whose channel handle is in `active.pcb`.

Sets `active.isactive` to TRUE and sets the `FCASY_WRITE_QUEUED` flag in `fcasy.flags` to indicate a write request.

Since the write is asynchronous, the memory pointed to by `buf` and `plen` must be preserved until the request completes.

AO_RUN**Process read or write completion**

```
INT ao_run(VOID);
```

Process the completion of either a read or a write request.

Note that the object is expected to handle the completion of either a read request (AO_QUEUE) or a write request (FC_WRITE) but not a mixture of the two.

Clears the FCASY_READ_QUEUED and FCASY_WRITE_QUEUED bits in `fcasy.flags`.

If `active.stat` is `TRUE` (indicating an error), it sends itself an `AO_CANCEL` message. This closes the file on detection of a read or write error (read errors include `E_FILE_EOF`); in the case of a write it ensures that the partially written file is deleted.

Sends itself an `FC_REQUEST_COMP` message before returning `RUN_ACTIVE_USED`.

★ FA_CLOSE**Close the file**

```
VOID fa_close(VOID);
```

Close the file by supersending an `FA_CLOSE` message. Sets `fcasy.recvname` (which is used on detection of an error to delete any partially written file) to `NULL`.

★ FC_OPEN**Open a file**

```
INT fc_open(TEXT *name, INT mode, FCOPY_DISP *pdinfo);
```

Open the file with `name` in the buffer pointed to by `name` (which must be at least `P_FNAME_SIZE` bytes long) in the specified `mode` and fill in the struct at `pdinfo` with as much information as possible about the file. The items that have been written to `*pdinfo` are indicated by the corresponding flag bits being set in `pdinfo->flags`.

It is assumed that the file is to be opened for either reading or writing, but not both. The value of `mode` must include one of `P_FOPEN`, `P_FCREATE` or `P_FREPLACE` (otherwise the method returns `E_GEN_ARG`).

Sets `pdinfo->sfname` to `name` and then parses `name` (using `p_fparse`) to ensure that the file name is valid. If the file is opened for writing and the specified directory does not exist, it is created automatically.

If the file is to be opened for reading (`mode` includes `P_FOPEN`), the file size and last modification date are written to `pdinfo->sfsz` and `pdinfo->sfdte` respectively. Depending on whether the file type is binary or text, `pdinfo->sftype` is set to `P_FSTREAM` or `P_FTEXT`. The value of `mode` is augmented by `ORing` in `P_FSHARE`. If the file type is text, `mode` is converted to use `P_FSTREAM_TEXT` (rather than `P_FTEXT`) to optimise the reading of the file.

If the file is to be opened for writing the method will return an error if `mode` includes `P_FCREATE` and the file exists or if `mode` includes `P_FREPLACE` and `name` specifies a directory file. Otherwise `pdinfo->rfname` is set to `name` and the value of `mode` is augmented by `ORing` in `P_FUPDATE`. If the file is opened successfully `fcasy.recvname` is set to `name`. The data space pointed to by `name` must therefore be preserved until the file has been closed.

Returns zero if successful, otherwise a negative error number.

Deferred FCASY methods

FC_REQUEST_COMP

Inform of completion

```
VOID fc_request_comp(VOID);
```

A deferred method, called from within the `ao_run` method, indicating that the read or write request has completed.

If there was a read or write error, the file will already have been closed and, if appropriate, deleted by means of an `AO_CANCEL` message. The subclass is, however, responsible for checking the completion status (`active.stat`) and performing any additional error handling.

In the absence of such errors, the user is responsible for continuing the operation by sending the next `AO_QUEUE` or `FC_WRITE` message.

Errors in this method should result in `p_leave` being called.

FCSYNC

ACTIVE	FACTIVE	FCASY	FCSYNC
q priority isactive pcb stat	owner	recvname flags	
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	~~fa_close~~ ao_init ~~ao_cancel~~ ao_abrun	ao_cancel ao_run ~~ao_queue~~ fa_close ~~fc_write~~ fc_open <i>fc_request_comp</i>	ao_queue fc_write

The `FCSYNC` abstract class subclasses `FCASY`. It converts `FCASY` to use synchronous file read and write services, otherwise it is identical to `FCASY`.

As with `FCASY`, it is assumed that an instance of a subclass of `FCSYNC` is used to either read from or write to a file.

Since file access is synchronous, it should only be used in situations where the read and write operations are guaranteed to complete quickly. Thus, it is effectively restricted to use with files on the local filing system.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```
CLASS fcsync fcasyn
Synchronous file I/O for local file system
{
  REPLACE ao_queue      Synchronous file read
  REPLACE fc_write      Synchronous file write
}
```

Property

None.

FCSYNC methods

★ AO_QUEUE

Read request

```
INT ao_queue(UBYTE *buf, UWORD *plen);
```

Perform a synchronous read of `*plen` bytes into `buf` from the file whose channel handle is in `active.pcb` by supersending the `AO_QUEUE` message and then waiting (with `p_waitstat`) on `active.stat` for the read to complete.

Sends itself an `AO_CANCEL` (which closes the file and sets `active.stat` to `E_FILE_CANCEL`) if the read completes with an error. In particular, since `E_FILE_EOF` is not distinguished from other errors, the file will be closed automatically on reading to the end of the file.

On exit `active.isactive` and `fcasy.flags` are both guaranteed to be zero.

Returns the value of `active.stat`.

★ FC_WRITE

Write request

```
INT fc_write(UBYTE *buf, UINT len);
```

Perform a synchronous write of `len` bytes from `buf` to the file whose channel handle is in `active.pcb` by supersending the `FC_WRITE` message and then waiting (with `p_waitstat`) on `active.stat` for the write to complete.

Sends itself an `AO_CANCEL` (which closes and deletes the file, and sets `active.stat` to `E_FILE_CANCEL`) if the write completes with an error.

On exit `active.isactive` and `fcasy.flags` are both guaranteed to be zero.

Returns the value of `active.stat`.

CHAPTER 15

FILE LISTS

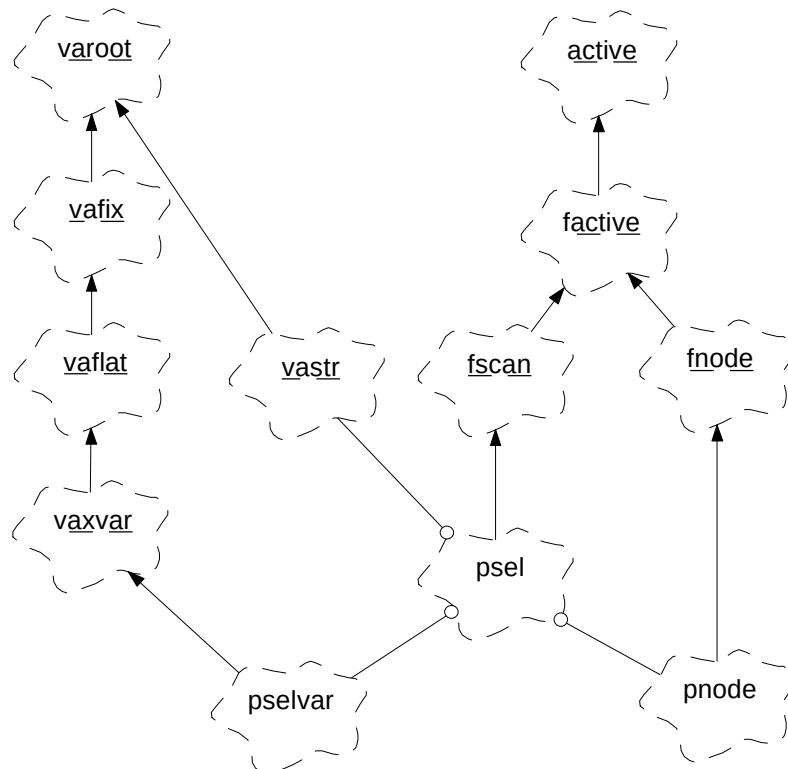
The classes described in this chapter are concerned with the navigation of the directories of a filing system and with the generation and storage of directory and file name lists.

Precursors

The reader is assumed to understand:

- active objects and the FNODE and FSCAN abstract classes
- the VASTR and VAXVAR variable array classes
- the file server node, device, directory and file services
- the p_enter and p_leave error handling services

Class diagram



PSELVAR

VAROOT		VAFIX	VAFLAT	VAXVAR	PSELVAR
nrec	key	rlen	gran nspc base		
destroy	~~va_replace~~	~~va_replace~~	~~va_init~~	~~va_test~~	~~va_test~~
va_count		~~va_copy~~	va_compress	va_copy	
va_delete	~~va_copy~~	~~va_reclen~~	~~va_deletem~~	va_reclen	
va_sort	~~va_reclen~~	va_swap	~~va_insertm~~	va_replace	
va_key	~~va_swap~~		va_capacity	va_init	
va_findisq	~~va_init~~		va_prec	va_deletem	
va_insertisq	~~va_deletem~~		~~va_pbuf~~	va_insertm	
va_append	~~va_insertm~~			va_pbuf	
va_insert	~~va_prec~~				
va_search	~~va_pbuf~~				
va_compare	~~va_capacity~~				
va_reset	~~va_compress~~				
va_test					

The PSELVAR class subclasses the VAXVAR variable array class. It is intended to be used to store an ordered array of file and directory names and the additional method is tailored to the special sorting schemes needed.

This class is defined specifically for use by the PSEL class, described later. The `va_test` method assumes that `varoot.key.desc` specifies ascending or descending order as normal for the variable array classes, but that `varoot.key.fold` contains one of the PSEL_ORDER_XXX values defined below. The `varoot.key ofs` and `varoot.key.len` fields are not used.

The PSEL `ps_order` method writes directly to the `varoot.key.desc` and `varoot.key.fold` fields of its component instance of PSELVAR (as an alternative to providing PSELVAR with a subclassed `va_key` method).

Each PSELVAR record is assumed to be an RC_VAXVAR struct, whose `buf` field points to an allocated heap cell containing a PSEL_REC struct. The contents of this struct are determined by the owning PSEL. The `name` field contains either a file name or a directory name. Note that the `namlen` field contains meaningful data only if the names are ordered by extension.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```

CLASS    pselvar vxaxvar
Holds the files/dir names in order - dir names first then file names
{
  REPLACE va_test
  CONSTANTS
  {
    PSEL_FLAG_TAG    0x1

    PSEL_ORDER_NAME      0      Order by name alphabetically (default)
    PSEL_ORDER_TIME      1      Order by time of creation
    PSEL_ORDER_DATE      2      Order by date of creation
    PSEL_ORDER_SIZE      3      Order by size of file
    PSEL_ORDER_EXT       4      Order by extension
  }
  TYPES
  {
    typedef struct
    {
      UWORD flags;           Tagging flags info
      UWORD namlen;         Offset in name of extension
      P_INFO info;          File info
      UBYTE name[P_FNAME_SIZE]; Max file name size buffer
    } PSEL_REC;
  }
}

```

Property

None.

PSELVAR methods

★ VA_TEST

Compare two records by pointer

```
INT va_test(RC_VAXVAR *prec1, RC_VAXVAR *prec2);
```

Compare the two records pointed to by `prec1` and `prec2`, returning 0 if the two records are equal, <0 if *`prec1` is before *`prec2`, or >0 if *`prec1` is after *`prec2`.

Each record is assumed to be an `RC_VAXVAR` struct whose `buf` field points to a `PSEL_REC` struct containing either a file name or a directory name.

A directory name is always ordered before a file name and directory names are always ordered alphabetically.

File names are ordered by one of name, creation time, creation date, size or extension, depending on the `PSEL_ORDER_XXX` value stored in `varoot.key.fold`.

The result of the comparison is reversed if `varoot.key.desc` is non-zero.

PNODE

ACTIVE	FACTIVE	FNODE	PNODE
q priority isactive pcb stat	owner	flags pname oldname pcb	
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	~~fa_close~~ ao_init ao_cancel ~~ao_abrun~~	fn_list ao_queue ao_run fa_close ~~fn_end_list~~ ~~fn_nodename~~	ao_abrun fn_end_list fn_nodename

The `PNODE` class subclasses the `FNODE` node and device list generator.

It is defined specifically for use by the `PSEL` class, described later. The code contains assumptions that the owner (whose handle is in `factive.owner`) is an instance of a subclass of `PSEL` and makes direct calls to `PSEL` code.

The `PNODE` methods are designed only to be called via the mechanisms provided within the methods of the `PSEL` class, described later in this chapter.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```
CLASS pnode fnode
Used by psel to generate the node/directory array
{
  REPLACE ao_abrun
  REPLACE fn_end_list
  REPLACE fn_nodename
}
```

Property

None.

PNODE methods

AO_ABRUN

Handle error

```
VOID ao_abrun(VOID);
```

Supersend the AO_ABRUN message and then make a direct call to reset property and component objects of the owner (assumed PSEL).

FN_END_LIST

Handle completed list

```
VOID fn_end_list(VOID);
```

Process the completion of the *node::device:*\ name list.

Reads and modifies the property of the owning (PSEL) object to indicate that the list has been changed, and starts the generation of the file list corresponding to the selected, or the default, item in the *node::device:*\ list.

FN_NODENAME

Process a list item

```
VOID fn_nodename(VOID);
```

Process a generated *node::device:*\ name.

Makes a direct call to add the generated name to the owner's (PSEL) directory/node name list and then sends itself an AO_QUEUE message to continue the scan.

PSEL

ACTIVE	FACTIVE	FSCAN	PSEL
q priority isactive pcb stat	owner	flags index pname match delim finfo crk pcbarr name	pfile pdir pnode dirent flags ascnt dirnum setpath builderr fck fspec
destroy ao_init ao_cancel ao_abrun ao_queue ao_run	fa_close ao_init ao_cancel ao_abrun	fs_matchname fs_fscan ao_queue ao_run fa_close ~~fs_fscan_end~~ ~~fs_dirname~~ ~~fs_filename~~ fs_end_dirlist	ao_init ao_abrun ao_cancel fs_filename fs_dirname fs_fscan_end ps_get_file ps_ascend_path ps_descend_path ps_set_path ps_sense_filename ps_select_dirent ps_drives ps_settag ps_gettag ps_order <i>ps_new_list</i>

PSEL is an abstract subclass of FSCAN, providing methods to navigate a filing system and to generate both a node list and a file name list from a wildcard file specification. In addition it provides methods to tag items in its file name list and to retrieve such tagged items.

The node list contains device names or directory names at some specified level. The file name list contains files and subdirectories within one particular item in the node list.

Although PSEL uses a number of active object components, it appears to a user as a single active object. A queued request starts the building of one or more of its lists. On completion it reports:

- which lists have been changed
- whether the file specification has changed
- whether the file name list was either built successfully or was left empty because the physical device does not exist (no disk in drive).

A subclass of PSEL is used, for example, to generate and navigate the file lists displayed in the file commands of the Series 3 System application, and in the MC File Manager application.

Subclasses of PSEL are not expected to replace any of the supplied methods.

Note that there is no automatic detection of filing system changes. This means that a file list will be out of date if, for example, another process has created a file in the directory being displayed after the file list was last built. A regeneration of the file name list may be forced by sending a PS_SET_PATH message.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```

CLASS    psel    fscan
Builds a node/directory array and a file array from a wild card path
{
  REPLACE ao_init
  REPLACE ao_abrun
  REPLACE ao_cancel
  REPLACE fs_filename
  REPLACE fs_dirname
  REPLACE fs_fscan_end
  ADD ps_get_file
  ADD ps_ascend_path
  ADD ps_descend_path
  ADD ps_set_path
  ADD ps_sense_filename
  ADD ps_select_direntry
  ADD ps_drives
  ADD ps_settag
  ADD ps_gettag
  ADD ps_order
  DEFER ps_new_list
  CONSTANTS
  {
    PSEL_RESET_DIR_ARRAY    0x01    True if NOT reset node/dir array
    PSEL_RESET_FILE_ARRAY   0x02    True if NOT reset files array
    PSEL_RESET_FSPEC        0x04    True if NOT reset fspec[]
    PSEL_DEVICE_ARRAY       0x10    Building device list
    PSEL_DIR_ARRAY          0x20    Building dir list
    PSEL_FILE_ARRAY         0x40    Building file list
    PSEL_DESCEND            0x80    Building due to a descend
    PSEL_ASCEND             0x100    Building due to an ascend
    PSEL_SETPATH            0x200    Building due to a set path
    PSEL_INIT               0x400    Building due to an init
    PSEL_SELDIR             0x800    Building due to select dir
    PSEL_ANOTHER_CMD        0x1000
    PSEL_GENERATE_DEF       0x2000
    PSEL_QUEUED_CMD         0x4000    Running a queued cmd
    PSEL_CURRENTLY_BUSY     (0x100|0x80|0x400|0x200|0x800)
    PSEL_SET_TAG            -1
    PSEL_CLEAR_TAG          0
    PSEL_TOGGLE_TAG         1
  }
}

```

```
PROPERTY 3
{
    PR_PSELVAR *pfile;           File name lists
    PR_VASTR *pdir;             Directory/Node name list
    PR_PNODE *pnode;           Node List Generator
    UWORD dirent;              Which directory in list should be highlighted
    UWORD flags;               File selector control flags
    UWORD ascnt;
    UWORD dirnum;
    UBYTE *setpath;
    WORD builderr;             Files list build error to report
    P_FPARSE fck;              Currently cracked wildcard file spec
    TEXT fspec[P_FNAME_SIZE];  Current wildcarded file spec
}
```

Property

<code>psel.pfile</code>	the handle of an instance of PSELVAR, which holds the file name list in an array of PSEL_REC records. A subclass may access this handle, and will typically pass it to display code. (Note that PSELVAR's VA_PBUF method returns a pointer to a PSEL_REC struct, rather than to the text of a file name.)
<code>psel.pdir</code>	the handle of an instance of VASTR, which holds the array of names for a node or directory list. A subclass may access this handle and will typically pass it to display code.
<code>psel.pnode</code>	the handle of an instance of PNODE, which generates a <i>node::device:</i> name list. It should not be accessed by a subclass.
<code>psel.dirent</code>	the index of the current node list item in the <code>psel.pdir</code> VASTR array. It defines the subdirectory for which the file list is built (a value of -1 indicates that no entry is current). A subclass may read this field to indicate which displayed node list entry to highlight.
<code>psel.flags</code>	controlling flags used to drive the PSEL object. A subclass may read the values of the PSEL_RESET_DIR_ARRAY, PSEL_RESET_FILE_ARRAY and PSEL_RESET_FSPEC bit-fields.
<code>psel.ascnt</code>	the outstanding number of directory ascends to be performed (a user may request multiple ascends faster than they can be serviced). It should not be accessed by a subclass.
<code>psel.dirnum</code>	the index of the most recently selected entry from the <code>psel.pdir</code> VASTR array. Typically the user interface will allow the selection of a new entry from this array while PSEL is busy building the file names array for a previously selected entry. It should not be accessed by a subclass.
<code>psel.setpath</code>	a pointer to the most recently selected wildcarded file name specification from which the node and file name arrays are to be built. The data space to which it points must be preserved until the arrays have been fully built. It should not be accessed by a subclass.
<code>psel.builderr</code>	an error number corresponding to a drastic error (such as out of memory, or the removal of a filing system) which occurred while building the node and file name arrays. If no error has occurred the value will be zero. A subclass is expected to test this value to determine the build completion result when the deferred PS_NEW_LIST method is called.
<code>psel.fck</code>	the parsed file name information for the wildcarded name (contained in <code>psel.fspec</code>) that is currently being used to build the node and file name arrays. A subclass should treat this as a read-only field.
<code>psel.fspec</code>	the wildcarded file name specification that drives the generation of the node and file name arrays. Typically this field could be used to display the full file name specification corresponding to the arrays that have just been built. A subclass should treat this as a read-only field.

PSEL methods

A user is expected to send only those explicit messages that appear in the following list:

PS_ASCEND_PATH	ascend one subdirectory level
PS_DESCEND_PATH	descend to a subdirectory
PS_SET_PATH	set a new path
PS_SENSE_FILENAME	sense a file list item
PS_SELECT_DIRENTRY	select a node list entry
PS_DRIVES	ascend to the drives level
PS_SETTAG	set/clear a file tag
PS_GETTAG	get a tagged file
PS_ORDER	set the file list order

The remaining methods are intended for internal use.

Many of the above methods cause either or both of the directory/node and file name lists to be rebuilt. If this fails (say, because the pack has been removed or the filing system no longer exists) then the lists are rebuilt at the *node::device:* level. All such methods result in a (deferred) PS_NEW_LIST message being sent. Depending on whether rebuilding is required, this message may be sent either before the method returns or at some later time, when building is complete.

AO_INIT

Initialise

```
VOID ao_init(TEXT *pname);
```

Supersends an AO_INIT message to add itself to the APPMAN queue and then creates and initialises the *psel.pdir*, *psel.pfile* (with a granularity of 32) and *psel.pnode* components.

Then starts building the node and file name arrays, using the text pointed to by *pname* as the initial file name specification.

If the initial file specification is not valid (say, because the filing system no longer exists) the default lists are built. These consist of:

- a node list containing all currently available *node::device:* names
- a file name list for the directory indicated by the first item in the node list, containing all files that match the initial file name specification.

Calls *p_leave* on error (out of memory).

AO_ABRUN

Handle error

```
VOID ao_abrun(VOID);
```

Handle an error arising during the processing of an AO_RUN message.

Discards the contents of the node and file name lists (at least one of which is likely to be only partially built) and sets *psel.fspect* to contain a null string.

Sends a PS_NEW_LIST message, which will normally cause the user interface to display empty lists and then supersends the AO_ABRUN message.

This method will only be invoked by out of memory errors or by the disappearance of filing systems during the generation of the lists.

★ AO_CANCEL

Cancel list building

```
VOID ao_cancel(VOID);
```

Sends an AO_CANCEL to *psel.pnode* to cancel any request on the node list generator and then supersends an AO_CANCEL to cancel any file scan request. These two messages ensure that any open channels are closed. Following this, the contents of the node and file name lists are discarded.

This method is intended only to be executed as a consequence of PSEL receiving a DESTROY message. A user should not, for example, send an AO_CANCEL message when requesting an operation (such as an ascend) before a previous operation has completed. Such a sequence is handled by PSEL's internal logic and the user should simply request the new operation.

FS_FILENAME**Add a file name**

```
VOID fs_filename(VOID);
```

Add a file name to the file name list.

Generates a PSEL_REC for the file name pointed to by `fscan.pname` and appends it to the file name list (the list will be ordered when it is complete).

Note that, to minimise memory use, the PSEL_REC struct is adjusted to the exact length required for its content before it is appended. You should also be aware that the contents of the `namlen` field of the PSEL_REC struct will only be valid if the file name list is sorted by extension.

If the name is added successfully, sends an AO_QUEUE message to continue the scan.

May call `p_leave(E_GEN_NOMEMORY)`.

FS_DIRNAME**Add a directory name**

```
VOID fs_dirname(VOID);
```

Add a directory name to either the node list or the file name list depending on which list is currently being built.

If the name is being added to the node list, the full name (taken from the start of the `fscan.name` buffer) is inserted in alphabetical order.

If it is being added to the file name list, only the 'file name' part (pointed to by `fscan.pname`) is appended to the list exactly as described for the `fs_filename` method.

If the name is added successfully, sends an AO_QUEUE message to continue the scan.

May call `p_leave(E_GEN_NOMEMORY)`.

FS_FSCAN_END**Process end of a scan**

```
VOID fs_fscan_end(VOID);
```

Process the end of the building of either the node list or the file name list.

If the node list has just been built, the building of the appropriate file name list is started.

If the node list has not changed (say, because the build was initiated by a directory ascend request when already at the drives level), the processing is as for the completion of the building of the file name list, described below.

If the file name list has just been built, a check is made to see if any additional requests (such as one or more directory ascends) have been made while the list was being built. If so, the appropriate list building is restarted. Otherwise, provided the file name list has changed, it is sorted in the currently specified order and a PS_NEW_LIST message is sent to indicate that list building is complete.

PS_GET_FILE**Build file name list**

```
INT ps_get_file(VOID);
```

Discard the current file name list and send an FS_FSCAN message to start a scan to build a new list for the directory specified by the current item in the node list. The list will include all subdirectories, together with all file names that match the wildcard file name and extension string contained within the full file specification in the `psel.fspec` buffer.

Returns zero, indicating a successful start of the scan.

Calls `p_leave` on error.

★ PS_ASCEND_PATH**Ascend one subdirectory level**

```
VOID ps_ascend_path(VOID);
```

Start the rebuilding of the node and file name lists for a directory level one higher than that specified by the wildcarded full file specification in `pse1.fspec`.

If the current directory level is at the `node::device:\` level already (if, for example, `pse1.fspec` contains "LOC::A:*.IMG") then no further ascends can be made. The node list will, however, be re-built because a filing system may have been added or removed since the last time the lists were built. This is the only way in which such a change in the filing system can be recorded in the node list.

In such a case the files list will not be re-built unless the current node has disappeared.

If this message is received while the lists are in process of being built then the ascend request will be stored internally. On completion of the current build, building will be restarted (without sending a `PS_NEW_LIST` message) at the new directory level.

★ PS_DESCEND_PATH**Descend to a subdirectory**

```
VOID ps_descend_path(UINT entryno);
```

Descend into the subdirectory specified by record number `entryno` in the file name list and start the rebuilding of the node and file name lists.

Does nothing if the lists are currently being built (since the array from which the entry was selected no longer exists).

An `entryno` of -1 (meaning that no list item was selected) has the same effect as a value of 0, specifying the first item.

The file name list is checked to ensure that it contains the entry number specified and that the corresponding record is the name of a directory. If either of these tests fail the method does nothing.

Otherwise the node and file name lists are rebuilt for the new directory level.

★ PS_SET_PATH**Set a new path**

```
VOID ps_set_path(TEXT *pname);
```

Start the rebuilding of the node and file name lists to correspond with the wildcard full file specification pointed to by `pname`.

If the lists are currently being built, the new path name pointer is stored until the appropriate time that the current build can be abandoned and a new build started.

In all cases the data space pointed to by `pname` must be preserved until a `PS_NEW_LIST` message is received to indicate that the lists have been rebuilt.

★ PS_SENSE_FILENAME**Sense a file list item**

```
INT ps_sense_filename(INT entryno, PSEL_REC **pprec);
```

Write, to `*pprec`, a pointer to the `PSEL_REC` data for the file list item with record number `entryno`.

Returns zero if successful. Does not write to `*pprec` and returns `E_FILE_LOCKED` if the list is currently being built.

★ PS_SELECT_DIRENTRY**Select a node list entry**

```
VOID ps_select_direntry(INT entryno);
```

Start the building of the file name list for the directory specified by item number `entryno` in the node list.

If a file name list is currently being built as a result of an earlier `PS_SELECT_DIRENTRY` message, the current activity is aborted and the build is restarted for the new list. This provides a rapid response to repeated `PS_SELECT_DIRENTRY` messages received from the user interface (generated, for example, as the user moves a highlight up and down a displayed list). If a list is being built for any other reason, the `PS_SELECT_DIRENTRY` message has no effect.

★ PS_DRIVES**Ascend to the drives level**

```
VOID ps_drives(VOID);
```

Start the rebuilding of the node and file name lists to generate a node list containing *node::device:* names and a file name list containing those items which match the file name and extension contained in the wildcarded full file specification in the *psel.fspect* buffer.

Does nothing if the lists are currently being built.

If the lists are currently at the drives level then they are not rebuilt.

In either case a *PS_NEW_LIST* message will be sent to indicate that the list building is complete.

★ PS_SETTAG**Set/clear a file tag**

```
INT ps_settag(INT entryno, INT flag);
```

Set, clear or toggle the tag status (held in the *flags* field of the *PSEL_REC* struct) of the file name list item specified by *entryno*.

The tag status will be set if *flag* is *PSEL_SET_TAG*, cleared if *flag* is *PSEL_CLEAR_TAG* and toggled if *flag* is *PSEL_TOGGLE_TAG*.

Returns *TRUE* if the tag status has been set, and *FALSE* if it has been cleared.

★ PS_GETTAG**Get a tagged file**

```
INT ps_gettag(INT index, PSEL_REC **pprec);
```

Write, to **pprec*, a pointer to the *PSEL_REC* of an item in the file name list that has its tag status set, and return either a (positive) value to be used as the *index* for a subsequent *PS_GETTAG* message, or *E_FILE_EOF* if there are no further tagged entries.

If *index* is zero, the value written to **pprec* points to the first tagged item. If *index* is the value returned by the previous *PS_GETTAG* the value written to **pprec* is a pointer to the next tagged item.

This method allows the extraction of the names of all tagged files, one by one.

PS_ORDER**Set the file list order**

```
VOID ps_order(INT mode, INT reverse);
```

Set the current file name list order and sort the entries. The value of *mode* should be one of:

<i>PSEL_ORDER_NAME</i>	order alphabetically by full name (the default)
<i>PSEL_ORDER_TIME</i>	order by time of creation
<i>PSEL_ORDER_DATE</i>	order by date of creation
<i>PSEL_ORDER_SIZE</i>	order by size of file
<i>PSEL_ORDER_EXT</i>	order alphabetically by extension

If *reverse* is *TRUE* the ordering is reversed.

The current file name list is regenerated in the new order. If the ordering is not by extension the reordering will be completed before this method returns. Otherwise the method starts a build of the file name list in the specified order and this will complete at some future time. In either case an *PS_NEW_LIST* message is sent when the list is complete.

All subsequent builds of the file name list will be performed in the specified order, until it is changed by a further *PS_ORDER* message.

By default the file name list is built in ascending alphabetical name order.

Deferred PSEL methods

★ PS_NEW_LIST Process the completion of list building

```
VOID ps_new_list(VOID);
```

A deferred method which is *always* received on completion of any request to build or reorder the node and/or file name arrays. It may be received either before the build request returns or at a later time, depending on whether any lists need to be rebuilt.

The supplier of this method is expected to test `psel.blderr` to determine the result of the build, and may also read `psel.flags`, `psel.pfile`, `psel.pdir`, `psel.dirent`, `psel.fck` and `psel.fspec`, as explained in the descriptions of the property fields.

A typical action would be to regenerate the data displayed in the user interface. It should examine `psel.flags` and:

- if `PSEL_RESET_DIR_ARRAY` is FALSE, regenerate the display of the directory/node names from the `psel.pdir` array
- if `PSEL_RESET_FILE_ARRAY` is FALSE, regenerate the display of the file names from the `psel.pfile` array
- if `PSEL_RESET_FSPEC` is FALSE, regenerate the display of the file name specification from the `psel.fspec` buffer

This method is expected to return. Any code that may result in `p_leave` being called must be run under the protection of `p_enter`.

CHAPTER 16

FILE MANAGEMENT CLASSES

The classes described in this chapter, and in particular the `FMAN` class, supply the basic engine for performing file management. They provide a set of high-level file system operations, for example, to copy a set of files, delete a directory structure or format an SSD.

`FMAN` creates and uses component instances of the `FMMK`, `FMFMT`, `FMSCAN`, `FMSRC` and `FMTARG` active object classes (which are all subclasses of `FACTIVE`). These components do the bulk of the work and, by breaking a potentially lengthy operation into small sections, ensure that the application remains responsive to window server events. Note that only one of these components is active at any one time.

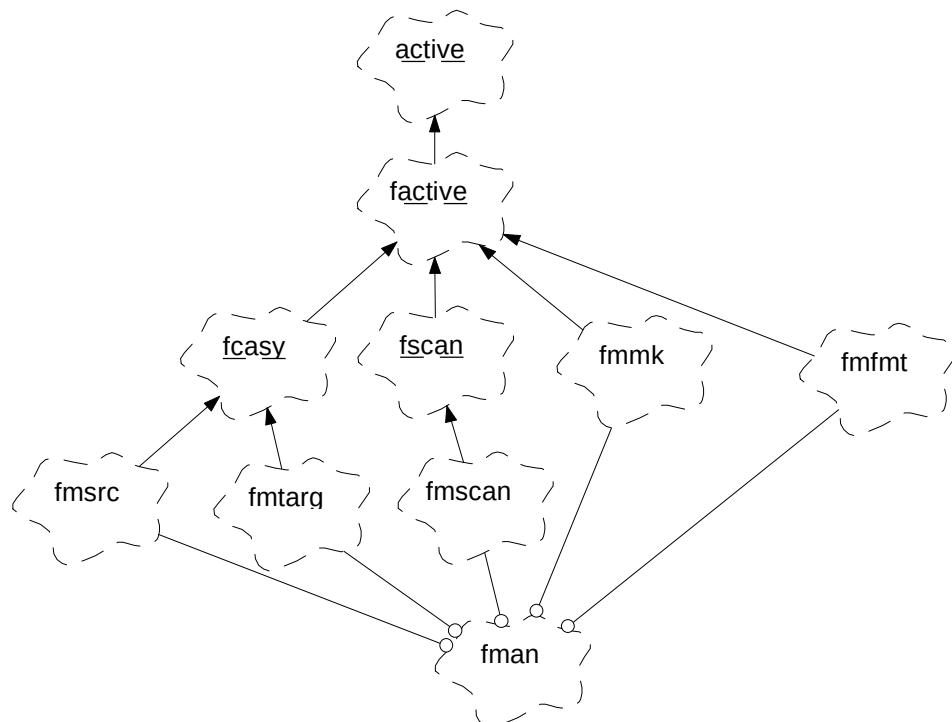
The classes are strongly interdependent. `FMAN` reads from and writes to the property of the other classes, which themselves rely on their being owned components of `FMAN`.

Precursors

The reader is assumed to understand:

- the active object scheduling mechanisms in the application manager
- the `FACTIVE`, `FSCAN` and `FCASY` classes
- the `PLIB` file system services
- the `p_enter` and `p_leave` error handling services

Class diagram



FMAN

FMAN	
srcfile	srclen
targfile	dispinfo
fmscan	targname
fmmk	wildsrcname
fmfmt	wildtargname
action	buf
mode	
fman_init	fman_name
fman_cancel	fman_info
fman_copy	fman_attrib
fman_delete	
fman_rename	<i>fman_complete</i>
fman_make	<i>fman_newname</i>
fman_remove	<i>fman_update</i>
fman_copydev	<i>fman_fileexist</i>
fman_format	<i>fman_error</i>

The FMAN (file manager) abstract class provides a set of file management operations. It must be subclassed to supply the deferred methods (which are chosen to provide a flexible interaction with any user interface) in order to create a useful object. It is not expected that a subclass will replace any of the supplied methods.

An FMAN operation is initiated by sending the appropriate message from the following list:

FMAN_COPY	copy files
FMAN_DELETE	delete files
FMAN_RENAME	rename files
FMAN_MAKE	make a directory
FMAN_REMOVE	remove a directory (and its subdirectories)
FMAN_COPYDEV	copy a device
FMAN_FORMAT	format a device
FMAN_NAME	name a device
FMAN_ATTRIB	set file attributes

Each of these methods will call `p_leave` on error. If successful in initiating the required action, some of these methods return zero, while others call `p_leave(0)` (to simplify the centralisation of error handling). It is therefore essential to send these messages under the protection of a `p_enter` (you could, for example, send the message by means of `p_entsend`). Under such protection the methods which call `p_leave(0)` will behave equivalently to those methods which return zero.

Any operation may complete before the send of the initialising message has returned. Normally, however, each of these operations will involve at least one active object (typically `FMSCAN`) which at some future time will be sent at least one `AO_RUN` message. Thus the operation will usually complete long after the send of the initiating message has returned.

Regardless of when it occurs, the completion may represent the successful conclusion of the operation, or may result from the operation being cancelled, either at the user's request or as the result of an error condition. Every completion, for whatever reason, results in the subclass of FMAN receiving one, and only one, `FMAN_COMPLETE` message. This is typically used to destroy any visual indicator of the current operation that is being presented to the user.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```

CLASS fman root
The File Manager
{
  ADD fman_init          Create/Init the active objects
  ADD fman_cancel        Cancel the last request
  ADD fman_copy          Copy files
  ADD fman_delete        Delete files
  ADD fman_rename        Rename files
  ADD fman_make          Make a directory
  ADD fman_remove        Remove a dir structure
  ADD fman_copydev        Copy a device
  ADD fman_format        Format a device
  ADD fman_name          Name a Pack
  ADD fman_info=p_dummy  For define of O_FMAN_INFO at least
  ADD fman_attrib        Set file attributes
  DEFER fman_complete    Operation now complete
  DEFER fman_newname      Operation now on this file(s)
  DEFER fman_update      Update copying display
  DEFER fman_fileexist    Destination file exists
  DEFER fman_error       Error in operation, abort/continue
CONSTANTS
{
  FMAN_READ_SIZE      0x800      Copy 2k at a time
  FMAN_COPYING        0
  FMAN_DELETE         1
  FMAN_RENAME         2
  FMAN_MAKE           3
  FMAN_REMOVE         4
  FMAN_FORMAT         5
  FMAN_NAME           6
  FMAN_ATTRIB         7
}

PROPERTY 5
{
  PR_FMSRC *srcfile;          Src/Read file active object
  PR_FMTARG *targfile;        Target/Write file active object
  PR_FMSCAN *fmSCAN;          File system scan active object
  PR_FMMK *fmmk;              Make Dir active object
  PR_FMFMT *fmfmt;            Format a pack.
  UWORD action;               Current file manager action
  UWORD mode;                 Current file create/replace mode
  UWORD srclen;
  FCOPY_DISP dispinfo;        Display info
  UBYTE targname[P_FNAME_SIZE]; Fully spec'ed target name
  UBYTE wildsrcname[P_FNAME_SIZE]; Entered wildcarded srcname
  UBYTE wildtargname[P_FNAME_SIZE]; Entered wildcarded target
  UBYTE buf[FMAN_READ_SIZE];
}
}

```

Property

<code>fman.srcfile</code>	the handle of an instance of FMSRC, representing the source file during a file copy. It should not be accessed by any subclass.
<code>fman.targfile</code>	the handle of an instance of FMTARG, representing the target file during a file copy. It should not be accessed by any subclass.
<code>fman.fmSCAN</code>	the handle of an instance of FMSCAN, used to generate the lists of files upon which an operation acts. It should not be accessed by any subclass.
<code>fman.fmmk</code>	the handle of an instance of FMMK, used to create a directory structure. It should not be accessed by any subclass.
<code>fman.fmfmt</code>	the handle of an instance of FMFMT, used to format an SSD. It should not be accessed by any subclass.

<code>fman.action</code>	the current file manager action, used to determine what operation to perform on a name generated by <code>FMSCAN</code>. It takes one of the following values: <code>FMAN_COPYING</code> - copying files <code>FMAN_DELETE</code> - deleting files <code>FMAN_RENAME</code> - we are renaming files <code>FMAN_ATTRIB</code> - changing the file attributes <code>FMAN_MAKE</code> - creating a directory structure <code>FMAN_REMOVE</code> - removing a directory structure <code>FMAN_FORMAT</code> - formatting an SSD <code>FMAN_NAME</code> - naming an SSD It should not be accessed by any subclass, but see also <code>fman.dispinfo</code>, which also contains this data.
<code>fman.mode</code>	the current create/replace mode for a file copy operation, At the start of a copy it is set to create the copy file(s) but may later be modified, depending on the result of an <code>FMAN_FILEEXIST</code> message. It should not be accessed by any subclass.
<code>fman.srclen</code>	how many bytes to read from a source file, and the length of buffered data to write to a target file. It should not be accessed by any subclass.
<code>fman.dispinfo</code>	information describing the current file manager action (see the <code>fman_newname</code> method for details of the <code>FCOPY_DISP</code> struct). It is intended to be used to provide information about the current operation for the user interface. It should only be read from within the <code>fman_newname</code> method.
<code>fman.targname</code>	the generated full file specification of a target file. It should not be accessed by any subclass.
<code>fman.wildsrcname</code>	the wildcarded file name passed as the source file name in one of the messages that initiates an operation. It should not be accessed by any subclass.
<code>fman.wildtargname</code>	the wildcarded file name passed as the target file name in one of the messages that initiates an operation between two files. It should not be accessed by any subclass.
<code>fman.buf</code>	the data read from the source file and written to the target file during a file copy. It should not be accessed by any subclass.

FMAN methods

FMAN_INIT Initialise the file manager

```
VOID fman_init(VOID);
```

Create and initialise the five active objects whose handles are stored in the first five items of `FMAN`'s property. The initialisation of each registers `FMAN` as its owner and adds the active object to the application manager's active object task queue.

Calls `p_leave(E_GEN_NOMEMORY)` on failure.

FMAN_CANCEL Cancel an outstanding request

```
VOID fman_cancel(VOID);
```

Cancel any current file operation, sending `AO_CANCEL` messages to the appropriate active object components, depending on the value of `fman.action`.

Following this, an `FMAN_COMPLETE` message is sent to indicate that the current operation has been completed.

FMAN_COPY**Copy files**

```
VOID fman_copy(UBYTE *src, UBYTE *targ, INT flags);
```

Start the copy of files specified by the wildcarded source file name at `src` to the wildcarded target name at `targ`.

Before the copy is started, the source and target file names are validated:

- The source file name must not be a null string and it must be a name acceptable to the `p_fparse` function (the name is parsed with "*" as the related name and the resulting full file specification is written into `fman.wildsrcname`).
- If the target name is a null string, it is replaced by the file name and extension taken from the full file specification in `fman.wildsrcname`. The target name is parsed with a NULL related name into `fman.wildtargname`. A final check is made that the resulting source and target file names are not identical.

If the validation generated the target name from the source name (because `targ` points to a null string) the name and extension in `fman.wildtargname` are replaced by "".

Following this, `fman.dispinfo.protocol` and `fman.action` are both set to `FMAN_COPYING` and `fman.dispinfo.blksiz` is set to `FMAN_READSIZE`, with `fman.dispinfo.flags` set to indicate that the appropriate fields are valid.

The value of `fman.mode` is set to `P_FCREATE|P_FSTREAM`, so that, initially, the copy files will be created (as opposed to, say, replacing existing files).

The `flags` parameter should contain a bitwise combination of one or more of the following values, defined in *factive.g*:

<code>FS_ALL_FILES</code>	include all non-directory files that are neither hidden nor system files
<code>FS_WRITABLE</code>	as <code>FS_ALL_FILES</code> , but excludes write-protected files (see below)
<code>FS_HIDDEN</code>	include hidden files
<code>FS_SYSTEM</code>	include system files
<code>FS_DIRECTORIES</code>	include directory files
<code>FS_MODIFIED</code>	exclude unmodified files (include only files that have <code>P_FAMOD</code> set)
<code>FS_INCLUDE_SUBDIRECTORIES</code>	extend the scan to subdirectories
<code>FS_PARSE_NAME</code>	parse file names before reporting them

The `FMSCAN` component has `fscan.flags` set to the passed `flags` value, ORED with `FS_HIDDEN|FS_SYSTEM`, and `fscan.match` is set to point to the name and extension within the `fman.wildsrcname` buffer. An `FS_FSCAN` message is then sent to the `FMSCAN` component to generate the list of files to copy. The initial start up file name passed with this message is the full contents of `fman.wildsrcname`.

Any error encountered during the file name validation or in `FMSCAN`'s `FS_FSCAN` method results in `p_leave` being called with an appropriate error.

On successful completion the method calls `p_leave(0)`. The `FMAN_COPY` message must therefore be sent under the protection of a `p_enter`.

Further processing of the file copy is handled by the `FMSCAN` component.

FMAN_DELETE**Delete files**

```
VOID fman_delete(UBYTE *src, INT flags);
```

Start the deletion of one or more files as specified by the wildcarded file specification string pointed to by `src`. In contrast to the `fman_remove` method, directories are not deleted.

Before the deletion is started the file specification string is validated. It must not be a null string and it must be a name acceptable to the `p_fparse` function (the name is parsed with "*" as the related name and the resulting full file specification is written into `fman.wildsrcname`).

Following this, `fman.dispinfo.protocol` and `fman.action` are both set to `FMAN_DELETE`. The value of `fman.dispinfo.flags` is set to indicate that the appropriate field is valid.

The `flags` parameter should contain a bitwise combination of one or more of the following values, defined in *active.g*:

<code>FS_ALL_FILES</code>	include all non-directory files that are neither hidden nor system files
<code>FS_WRITABLE</code>	as <code>FS_ALL_FILES</code> , but excludes write-protected files (see below)
<code>FS_HIDDEN</code>	include hidden files
<code>FS_SYSTEM</code>	include system files
<code>FS_DIRECTORIES</code>	include directory files
<code>FS_MODIFIED</code>	exclude unmodified files (include only files that have <code>P_FAMOD</code> set)
<code>FS_INCLUDE_SUBDIRECTORIES</code>	extend the scan to subdirectories
<code>FS_PARSE_NAME</code>	parse file names before reporting them

The FMSCAN component has `fscan.flags` set to the passed `flags` value, Ored with `FS_HIDDEN|FS_SYSTEM`, and `fscan.match` is set to point to the name and extension within the `fman.wildsrcname` buffer. An `FS_FSCAN` message is then sent to the FMSCAN component to generate the list of files to delete. The initial start up file name passed with this message is the full contents of `fman.wildsrcname`.

Any error encountered during the file name validation or in FMSCAN's `FS_FSCAN` method results in `p_leave` being called with an appropriate error.

On successful completion, the method calls `p_leave(0)`. The `FMAN_DELETE` message must therefore be sent under the protection of a `p_enter`.

Further processing of the file deletion is handled by the FMSCAN component.

FMAN_RENAME

Rename files

```
VOID fman_rename(UBYTE *src, UBYTE *targ);
```

Start the renaming of one or more files as specified by the wildcarded file specification string pointed to by `src` to names specified by the wildcarded file specification string pointed to by `targ`.

Before the rename is started the source and target file names are validated:

- The source file name must not be a null string and it must be a name acceptable to the `p_fparse` function (the name is parsed with "*" as the related name and the resulting full file specification is written into `fman.wildsrcname`).
- If the target name is a null string it is replaced by the file name and extension taken from the full file specification in `fman.wildsrcname`. The target name is parsed with a NULL related name into `fman.wildtargname`.

Final checks are made that the resulting source and target file names are not identical, and that both file specifications refer to the same directory (files can not be renamed across directories, devices or file systems).

If the validation generated the target name from the source name (because `targ` points to a null string), the name and extension in `fman.wildtargname` are replaced by "*".

Following this, `fman.dispinfo.protocol` and `fman.action` are both set to `FMAN_RENAME`. The value of `fman.dispinfo.flags` is set to indicate that the appropriate field is valid.

The FMSCAN component has `fscan.flags` set to `FS_ALL_FILES|FS_HIDDEN|FS_SYSTEM` and `fscan.match` is set to point to the name and extension within the `fman.wildsrcname` buffer. An `FS_FSCAN` message is then sent to the FMSCAN component to generate the list of files to rename. The initial start up file name passed with this message is the full contents of `fman.wildsrcname`.

Any error encountered during the file name validation or in FMSCAN's `FS_FSCAN` method results in `p_leave` being called with an appropriate error.

On successful completion the method calls `p_leave(0)`. The `FMAN_RENAME` message must therefore be sent under the protection of a `p_enter`.

Further processing of the file rename is handled by the FMSCAN component.

FMAN_MAKE**Make a directory tree**

```
INT fman_make(UBYTE *name);
```

Start the creation of a directory or directory structure as specified by name (using `p_mkdir`).

The values of `fman.dispinfo.protocol` and `fman.action` are both set to `FMAN_MAKE`. The string pointed to by name is copied into the `fman.targname` buffer and `fman.dispinfo.sfname` is set to point to this buffer. The value of `fman.dispinfo.flags` is set to indicate that the appropriate fields are valid.

Following this, the property of the FMMK component is modified directly, setting `active.isactive` to `TRUE`, and `active.stat` to zero. The I/O semaphore is then signalled (by calling `p_iosignal`) so that FMMK will, at some future time, receive an `AO_RUN` message.

The method returns zero.

See the FMMK class for details of further processing.

FMAN_REMOVE**Delete a directory structure**

```
INT fman_remove(UBYTE *name);
```

Remove the directory structure specified by the name of a directory pointed to by name. The specified directory and all included files and subdirectories are deleted.

The values of `fman.dispinfo.protocol` and `fman.action` are both set to `FMAN_REMOVE`. The value of `fman.dispinfo.flags` is set to indicate that the appropriate field is valid.

The passed name is parsed into the `fman.wildsrcname` buffer and adjusted, if necessary (with the aid of a call to `p_chdir`) to ensure that it contains a valid directory name for the relevant filing system. (For MSDOS, for example, it ensures that the directory name includes a trailing `\` character.) This may fail if the supplied text does not produce a valid directory name.

The FMSCAN component has `fmscan.match` set to point to the string `"*"` (to match all files) and `fmscan.flags` set to `FS_ALL_FILES|FS_INCLUDE_SUBDIRECTORIES|FS_HIDDEN|FS_SYSTEM` so that it will generate the names of all the files and directories below the specified directory. Following this, the FMSCAN component is sent an `FS_FSCAN` message. The initial start up file name passed with this message is the full contents of `fman.wildsrcname`.

Returns zero if the remove has been started successfully, or calls `p_leave` on error.

Further processing of the directory removal is handled by the FMSCAN component.

FMAN_COPYDEV**Copy a device**

```
VOID fman_copydev(UBYTE *src, UBYTE *targ, INT flags);
```

Initiate the copy of the device specified by `src` to the device and directory name specified by `targ`, reproducing the source device structure under the target directory. This method effectively provides a backup service. The results will be unpredictable if `src` does not point to a device name.

The value of `flags` should be either zero, to copy all files, or `FS_MODIFIED`, to restrict the copy to only those files which are marked as modified.

The values of `fman.dispinfo.protocol` and `fman.action` are both set to `FMAN_COPYING`, and `fman.dispinfo.blksiz` is set to `FMAN_READ_SIZE`. The value of `fman.dispinfo.flags` is set to indicate that the appropriate fields are valid.

The target name is copied into `fman.wildtargname`, and is converted to a directory name, as in the `fman_remove` method. The source name is parsed into `fman.wildsrcname` and both names are validated, as for the `fman_copy` method. A further check ensures that the source and target names do not specify the same device.

The value of `fman.mode` is set to `P_FREPLACE|P_FSTREAM`, so any previously existing target files will be overwritten without notification.

The FMSCAN component has `fscan.flags` set to the passed `flags` value, `ORED` with `FS_ALL_FILES|FS_INCLUDE_SUBDIRECTORIES|FS_HIDDEN|FS_SYSTEM`, and `fscan.match` is set to point to the string `"*"`. An `FS_FSCAN` message is then sent to the FMSCAN component to generate the list of files to copy. The initial start up file name passed with this message is the full contents of `fman.wildsrcname`.

Any error encountered during the name validation or in FMSCAN's `FS_FSCAN` method results in `p_leave` being called with an appropriate error.

On successful completion, the method calls `p_leave(0)`. The `FMAN_COPYDEV` message must, therefore, be sent under the protection of a `p_enter`.

Further processing of the device copy is handled by the `FMSCAN` component.

FMAN_FORMAT

Format a device

```
INT fman_format(UBYTE *devname, UBYTE *volname, UINT flags);
```

Initiate the formatting of the device specified by `devname`, giving it a volume name specified by `volname`.

Although there is no intrinsic restriction on the filing system in which the format is to take place, the *REM::* filing system currently does not support the required formatting services. Thus, an error condition will occur if an attempt is made to format a device on the *REM::* filing system.

The value of `flags` must be either zero or `P_FLOWDENSITY`. It specifies the format mode for the formatting of floppy disks. (This is primarily supplied to support future expansion since, at the time of writing, there is no *LOC::* floppy disk drive.)

The passed device name and volume name concatenated into `fman.targname` and the device is opened for formatting with the open file handle written directly to the `active.pcb` property of the `FMFMT` component.

The values of `fman.dispinfo.protocol` and `fman.action` are both set to `FMAN_FORMAT`, `fman.dispinfo.sfname` is set to point to the `fman.targname` buffer and `fman.dispinfo.blksiz` is set to 1. The value of `fman.dispinfo.flags` is set to indicate that the appropriate fields are valid.

The property of the `FMFMT` component is further modified, by setting `fmfmt.flags` to `FMFMT_NAME` (defined by the `FMFMT` class) `active.isactive` to `TRUE` and `active.stat` to zero. The I/O semaphore is signalled by a call to `p_iosignal`, ensuring that `FMFMT` will eventually receive an `AO_RUN` message.

Returns zero if the remove has been started successfully, or calls `p_leave` on error.

See the `FMFMT` class for details of further processing.

FMAN_NAME

Name a device

```
INT fman_name(UBYTE *devname, UBYTE *volname);
```

Change the volume name on the device specified by `devname` to the name pointed to by `volname`.

This method is exceptional, in that the action is performed synchronously. It will always have completed (either successfully or with an error) by the time the method terminates.

The method returns zero if the device was named successfully, or calls `p_leave` with the appropriate error number on failure.

FMAN_INFO

Obsolete method

```
VOID fman_info(VOID);
```

This method is no longer in use and is maintained solely to preserve the `OLIB` user interface.

FMAN_ATTRIB

Set file attributes

```
VOID fman_attrib(UBYTE *pname, UWORD attribs, UINT flags);
```

Initiate the setting of the file attributes specified by `attribs` for the set of files specified by the wildcarded name pointed to by `pname`.

The value of `attribs` may take any combination of the following flags:

<code>P_FAWRITE</code>	a writable file (not read only, deletable)
<code>P_FAHIDDEN</code>	a hidden file
<code>P_FASYSTEM</code>	a system file
<code>P_FAMOD</code>	the file is marked as modified.

Both the set and the clear states of all flags are significant; the clear state has the opposite meaning to the set state. For example, if the `P_FAWRITE` flag is not set then the file will be made read only.

The value of `flags` must be either `NULL` to restrict the operation to the single specified directory, or `FS_INCLUDE_SUBDIRECTORIES` to extend the setting of attributes to files within all subdirectories of the specified directory.

The values of `fman.dispinfo.protocol` and `fman.action` are both set to `FMAN_ATTRIB` and `fman.dispinfo.flags` is set to indicate that the appropriate field is valid.

The name pointed to by `pname` must not be a null string. Provided it passes this check, it is parsed (with a related name of `"*"`) into `fman.wildsrcname`, and `fman.mode` is set to the value of `attrs`.

The FMSCAN component has `fscan.flags` set to the passed `flags` value, ORed with `FS_ALL_FILES|FS_HIDDEN|FS_SYSTEM`, and `fscan.match` is set to point to the name and extension within the `fman.wildsrcname` buffer. An `FS_FSCAN` message is then sent to the FMSCAN component to generate the list of files whose attributes are to be changed. The initial start up file name passed with this message is the full contents of `fman.wildsrcname`.

Any error encountered during the name validation or in FMSCAN's `FS_FSCAN` method results in `p_leave` being called with an appropriate error.

On successful completion the method calls `p_leave(0)`. The `FMAN_ATTRIB` message must therefore be sent under the protection of a `p_enter`.

Further processing of the setting of attributes is handled by the FMSCAN component.

Deferred FMAN methods

★ FMAN_COMPLETE

Processing completed

```
VOID fman_complete(VOID);
```

Notify that an asynchronous file manager request has now completed.

This message will not be received if the request failed to start (if, for example, the `fman_copy` method called `p_leave` with a non-zero parameter). Provided the request started successfully, this message is guaranteed to be received, regardless of the reason for the completion. Such reasons include:

- there are no more files to handle
- the request was aborted by the user, or otherwise cancelled
- the request terminated with a non-recoverable error

Errors in the file manager request are handled (for example, via the `FMAN_ERROR` and `FMAN_FILEEXIST` messages) prior to receipt of this message and so the `fman_complete` method does not need an associated completion status.

Note that it is not considered an error if the requested action does not actually process any files, such as in a request to delete files from an empty directory. If such a condition needs to be reported, it may be detected by the receipt of an `FMAN_COMPLETE` message that is not preceded by one or more `FMAN_NEWNAME` messages.

Typically, a subclass would use the receipt of this message to destroy any objects associated with the display of status information for the current operation and to allow further file management actions be selected.

This method is expected to return. Any code that may result in `p_leave` being called must be run under the protection of `p_enter`.

★ FMAN_NEWNAME**Processing a new file**

```
VOID fman_newname(FCOPY_DISP *pinfo);
```

Notify that processing is about to commence on a new file.

Information about the file is contained in the `FCOPY_DISP` struct pointed to by `pinfo`. This struct is declared in the `FCASY` class, since it contains data that is used by a number of `FCASY` subclasses. For convenience it is reproduced here:

```
typedef struct
{
    UWORD flags;           Which fields are valid
    UWORD protocol;       Which protocol being used
    UWORD blksize;        Size of each block
    UWORD blkno;          Current block number
    UWORD sftype;         Source file type
    UWORD rftype;         Receive file type
    ULONG sfsz;           Source file size
    ULONG rfsz;           Receive file size
    ULONG sfdate;         Source file last modification date
    ULONG rfdate;         Receive file last modification date
    UBYTE *sfname;        Source file name
    UBYTE *rfname;        Receive file name
} FCOPY_DISP;
```

Since, in general, not all fields contain valid information, the `flags` field indicates which of the fields are valid as follows:

<code>FCOPY_DISP_SFTYPE</code>	<code>sftype</code> contains either <code>P_FTEXT</code> or <code>P_FSTREAM</code>
<code>FCOPY_DISP_SFSize</code>	<code>sfsz</code> contains the source file size, in bytes
<code>FCOPY_DISP_SFDATE</code>	<code>sfdate</code> contains, as a system time, the last modification date of the source file
<code>FCOPY_DISP_SFNAME</code>	<code>sfname</code> contains a pointer to the source file name as a zero terminated string
<code>FCOPY_DISP_RFTYPE</code>	<code>rftype</code> contains either <code>P_FTEXT</code> or <code>P_FSTREAM</code>
<code>FCOPY_DISP_RFSIZE</code>	<code>rfsz</code> contains the remote or target file size in bytes
<code>FCOPY_DISP_RFDATE</code>	<code>rfdate</code> contains, as a system time, the last modification date of the remote or target file
<code>FCOPY_DISP_RFNAME</code>	<code>rfname</code> contains a pointer to the remote or target file name as a zero terminated string
<code>FCOPY_DISP_BLKSIZE</code>	<code>blksize</code> contains a value that represents the amount of data that has been processed when each <code>FMAN_UPDATE</code> message is received
<code>FCOPY_DISP_BLKNO</code>	<code>blkno</code> contains a value specifying the current block number that is being processed (provided the appropriate fields are valid, the value of <code>sfsz/blksize</code> equals the maximum value of <code>blkno</code> that will be reached during the processing)
<code>FCOPY_DISP_PROTOCOL</code>	<code>protocol</code> contains a value specifying the current <code>FMAN</code> action (one of <code>FMAN_COPYING</code> , <code>FMAN_DELETE</code> , <code>FMAN_RENAME</code> , <code>FMAN_MAKE</code> , <code>FMAN_REMOVE</code> , <code>FMAN_FORMAT</code> , <code>FMAN_NAME</code> or <code>FMAN_ATTRIB</code>)

Irrespective of the contents of the `flags` field, *the contents should not be read outside the `fman_newname` method.*

An `FMAN_NEWNAME` message is sometimes sent even if an error has prevented one or more fields from being set up. It is therefore essential always to check if a field is valid before using its contents.

This method has two principal uses:

- it provides information about the current stage of processing that may be presented to the user
- it supplies a context for any particular error (for example, if a file is read only, it is known in advance that the `p_delete` service will fail with an `E_FILE_RDONLY` error).

This method is expected to return. Any code that may result in `p_leave` being called must be run under the protection of `p_enter`.

★ FMAN_UPDATE**Section of processing completed**

```
VOID fman_update(VOID);
```

Notify that the next section of the requested action has completed.

The message will only be received during the processing of the copy and format services, each time that either a block of `FMAN_READ_SIZE` bytes has been copied, or that a section of the pack has been formatted.

This method may be used to update any progress report that is being presented to the user. The values of `fman.dispinfo.sfsiz` and `fman.dispinfo.blksiz` can be used to determine the number of times that an `FMAN_UPDATE` message will be received, in order to report a percentage completion to the user. Note that when formatting (that is, when `fman.dispinfo.protocol` has the value `FMAN_FORMAT`) only the low 16 bits of `fman.dispinfo.sfsiz` contain valid data, so in this case the top 16 bits should be masked off.

This method is expected to return. Any code that may result in `p_leave` being called must be run under the protection of `p_enter`.

★ FMAN_FILEEXIST**File exists**

```
INT fman_fileexist(VOID);
```

Notify that the destination file of a copy exists.

This message will only be received only if `fman.mode` is not set to cause existing files to be replaced. It can therefore only be received if files are being copied as a result of an `FMAN_COPY` message (and not `FMAN_COPYDEV`) since the `fman_copydev` method sets `fman.mode` to include the `P_FREPLACE` flag.

The method is expected to return one of the following values:

- 1 (or any other negative value) abandon the copy service
- 0 replace this file only, further clashing names will cause this message to be received again
- 1 skip this file and continues with the next file
- 2 replace this file and all subsequent files, regardless of whether the destination file exists.

This method is expected to return. Any code that may result in `p_leave` being called must be run under the protection of `p_enter`.

★ FMAN_ERROR**Determine error response**

```
INT fman_error(INT errnum);
```

Determine the action to be taken in response to a detected error with error number `errnum`.

The method should return a value which specifies the form of error recovery. The range of options is different, depending on whether or not the current operation is a file copy (`fman.action` is `FMAN_COPYING`).

During a file copy the return value may be one of:

- 0 abandon copying the current file and continue with the following file
- 1 (or any non-zero value) abandon the entire copy operation.

During any other operation the return value may be one of:

- 0 retry the operation on the current file
- 1 abandon the entire operation
- 2 abandon the operation on the current file and continue with the following file.

This method is expected to return. Any code that may result in `p_leave` being called must be run under the protection of `p_enter`.

FMMK

ACTIVE	FACTIVE	FMMK
q priority isactive pcb stat	owner	
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ao_queue ~~ao_run~~	fa_close ao_init ao_cancel ao_abrun	ao_run

The FMMK class is designed to be used as a component of FMAN which uses it to implement the make directory service.

Since the supplied AO_RUN method contains assumptions about the property of the object whose handle is stored in `factive.owner`, it is not suitable for use other than as a component of FMAN.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```
CLASS fmmk factive
The 'Create directory' active object
{
  REPLACE ao_run
}
```

Property

None.

FMMK methods

AO_RUN Process a make directory request

```
INT ao_run(VOID);
```

Make the directory that was specified by an earlier FMAN_MAKE message.

Sends the owning FMAN an FMAN_NEWNAME message, passing the address of FMAN's `fman.dispinfo` struct.

Then makes the directory (and any necessary intermediate directories) with a call to `p_mkdir`. If this call returns an error, FMAN is sent an FMAN_ERROR message and, if this message returns FALSE, a further attempt is made to create the directory. This is repeated until either the directory creation succeeds or the FMAN_ERROR message returns a non-zero value.

Following this, FMAN is sent an FMAN_COMPLETE message to indicate that the processing is complete. Note that the `ao_run` method is executed only once for each FMAN_MAKE message.

The method returns RUN_ACTIVE_USED.

FMFMT

ACTIVE	FACTIVE	FMFMT
q priority isactive pcb stat	owner	rdword rdlen flags
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ao_queue ~~ao_run~~	fa_close ao_init ao_cancel ~~ao_abrun~~	ao_run ao_abrun

The FMFMT class is designed to be used as a component of FMAN, which uses it to implement the format service.

Since the supplied methods contain assumptions that the object whose handle is stored in `factive.owner` is an instance of the FMAN class, it is not suitable for use in other situations.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```

CLASS   fmfmt   factive
Format active object
{
  REPLACE ao_run
  REPLACE ao_abrun
  CONSTANTS
  {
    FMFMT_NAME      0x01
  }
  PROPERTY
  {
    UWORD rdword;
    UWORD rdlen;
    UWORD flags;           Controlling flags
  }
}

```

Property

<code>fmfmt.rdword</code>	a scratch buffer to receive data read during the format process.
<code>fmfmt.rdlen</code>	the number (two!) of bytes read into <code>fmfmt.rdword</code>
<code>fmfmt.flags</code>	controlling flag, initially set to <code>FMFMT_NAME</code> and cleared on the first pass through the <code>ao_run</code> method

FMFMT methods

AO_RUN

Process a format request

```
INT ao_run(VOID);
```

Perform a segment of the processing of a file manager format request.

If this is the first time that the method has been called following a `FMAN_FORMAT` message, the first read is made, to obtain the format count, and the result is written to the bottom 16 bits of FMAN's `fman.dispinfo.sfsz`. Following this, FMAN is sent an `FMAN_NEWNAME` message, passing the address of FMAN's `fman.dispinfo` struct.

On subsequent calls, unless an error condition is encountered, FMAN is sent an FMAN_UPDATE message, `active.stat` is set to TRUE and another read is queued.

If a read request completed with an E_FILE_EOF error, indicating that the format is complete, the method sends itself an FA_CLOSE message and then sends FMAN an FMAN_COMPLETE message.

All other errors result in `p_leave` being called.

The method returns RUN_ACTIVE_USED.

AO_ABRUN Process formatting error

```
VOID ao_abrun(VOID);
```

Process an error which caused the `ao_run` method to call `p_leave`.

Supersends an AO_ABRUN message and then sends FMAN an FMAN_COMPLETE message.

FMSCAN

ACTIVE	FACTIVE	FSCAN	FMSCAN
q priority isactive pcb stat	owner	flags index pname match delim finfo crk pcbarr name	
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	~~fa_close~~ ao_init ao_cancel ~~ao_abrun~~	fs_matchname fs_fscan ao_queue ao_run fa_close ~~fs_fscan_end~~ ~~fs_dirname~~ ~~fs_filename~~ ~~fs_end_dirlist~~	ao_abrun fs_filename fs_dirname fs_end_dirlist fs_fscan_end

The FMSCAN class is designed to be used as a component of FMAN, which uses it to implement the copy file, delete file, rename file, remove directory and copy device services.

Since the supplied methods contain assumptions that the object whose handle is stored in `factive.owner` is an instance of the FMAN class, it is not suitable for use in other situations.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```
CLASS fmscan fscan
File system scan active object
{
  REPLACE ao_abrun
  REPLACE fs_filename
  REPLACE fs_dirname
  REPLACE fs_end_dirlist
  REPLACE fs_fscan_end
}
```

Property

None.

FMSCAN methods

AO_ABRUN

Process an error

```
VOID ao_abrun(VOID);
```

Handle any error that arises within the `ao_run` method (provided by `FSCAN`).

Reads the owning `FMAN`'s property and, if `fman.action` is `FMAN_COPYING`, sends `AO_CANCEL` messages to the `fman.srcfile` and `fman.targfile` component objects.

Then supersends the `AO_ABRUN` message and sends `FMAN` an `FMAN_COMPLETE` message.

FS_FILENAME

Next file name

```
VOID fs_filename(VOID);
```

Process a new file name during a file name scan (the full file specification of the file is in the `fscan.name` buffer). This method will be called during a copy file, copy device, delete file, delete directory, rename file or set attribute service.

Reads the owning `FMAN`'s property and performs one of the following actions, depending on the value of `fman.action`:

<code>FMAN_COPYING</code>	the name for the copy file is generated by merging the current file name from the scan with the wildcarded target name (from <code>fman.wildtargname</code>). If this is successful, the source and target files are opened by sending <code>FC_OPEN</code> messages to <code>FMAN</code>'s <code>FMSRC</code> and <code>FMTARG</code> components. The source file is opened in read only mode (with the <code>P_FSHARE</code> flag set) and the target file in text (<code>P_FSTREAM_TEXT</code>) or binary mode, according to the source file type specified by <code>fman.mode</code>. Regardless of the success or failure of the name generation or the attempt to open the files, <code>FMAN</code> is sent an <code>FMAN_NEWNAME</code> message, passing the address of the <code>fman.dispinfo</code> struct (whose <code>sfname</code> and <code>rfname</code> elements point respectively to the source and target file names). This allows the user interface to display names that cause an error in addition to displaying valid names. The copy is started by sending an <code>AO_QUEUE</code> message to <code>FMAN</code>'s <code>FMSRC</code> component. Further processing is controlled by the <code>FMSRC</code> and <code>FMTARG</code> components.
<code>FMAN_RENAME</code>	the file's new name is generated by merging the current file name from the scan and the wildcarded target name (from <code>fman.wildtargname</code>) and <code>FMAN</code> is sent an <code>FMAN_NEWNAME</code> message, passing the address of the <code>fman.dispinfo</code> struct (whose <code>sfname</code> and <code>rfname</code> elements point respectively to the old and new file names). Provided the name generation did not fail, the file is renamed.
<code>FMAN_ATTRIB</code>	sends <code>FMAN</code> an <code>FMAN_NEWNAME</code> message and sets the file's attributes according to the value of <code>fman.mode</code>.
<code>FMAN_DELETE</code> or <code>FMAN_REMOVE</code>	sends <code>FMAN</code> an <code>FMAN_NEWNAME</code> message and deletes the file.

An `E_FILE_EXIST` error that occurs when trying to create the target file for a copy causes `FMAN` to be sent an `FMAN_FILEEXIST` message. See the earlier description of this method for the range of possible return values and the resulting actions.

All other errors cause an `FMAN_ERROR` message to be sent to `FMAN` and the return value determines what form of error recovery action is taken. The range of possible actions depends on the service that is being processed. The various options are listed in the description of the `fman_error` method.

Except when either copying files or, following an error, the user selects to abandon the entire operation, the scan is continued by sending `FSCAN` an `AO_QUEUE`.

FS_DIRNAME **Next directory name**

```
VOID fs_dirname(VOID);
```

Process a new directory file during a file name scan. This method will be called when scanning into a subdirectory during a copy or delete service.

Reads the owning FMAN's property and, if `fman.action` is `FMAN_COPYING`, adjusts the contents of the `fman.wildtargname` buffer to refer to the new subdirectory. Any error here results in `p_leave` being called.

Sends itself an `AO_QUEUE` message to continue the scan.

FS_END_DIRLIST **End of subdirectory**

```
VOID fs_end_dirlist(VOID);
```

Process the reaching of the end of a subdirectory during a file name scan.

Reads the owning FMAN's property and, if `fman.action` is `FMAN_REMOVE`, deletes the subdirectory whose name is specified in the `fman.name` buffer (any files contained in the subdirectory have been deleted earlier in the scan). Any error causes FMAN to be sent an `FMAN_ERROR` message. The return value determines the action as follows:

- 0 retry the deletion until it succeeds, or the user decides to skip or abandon
- 1 abandon the entire operation - send FMSCAN an `AO_CANCEL` message and FMAN an `FMAN_COMPLETE` message
- 2 continue, without deleting the subdirectory.

Otherwise, if `fman.action` is `FMAN_COPYING`, adjusts the contents of the `fman.wildtargname` buffer to refer to the parent directory. Any error here results in `p_leave` being called.

Sends itself an `AO_QUEUE` message to continue the scan *except* following a user decision to abort.

FS_FSCAN_END **Scan completion**

```
VOID fs_fscan_end(VOID);
```

Process the end of the file name scan.

Reads the owning FMAN's property and, if `fman.action` is `FMAN_REMOVE`, deletes the directory whose name is specified in the `fman.name` buffer (any files and subdirectories have been deleted during the scan). If `fman.name` specifies the root directory, the attempt to delete it will, of course, fail but this is not considered an error. Any other error causes FMAN to be sent an `FMAN_ERROR` message. The return value determines the action as follows:

- 0 retry the deletion until it succeeds, or the user decides to skip or abandon
- 1 abort the entire operation - send FMSCAN an `AO_CANCEL` message and send FMAN an `FMAN_COMPLETE` message
- 2 continue, without deleting the directory.

Sends FMAN an `FMAN_COMPLETE` message.

FMSRC

ACTIVE	FACTIVE	FCASY	FMSRC
q priority isactive pcb stat	owner	recvname flags	
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	~~fa_close~~ ao_init ~~ao_cancel~~ ao_abrun	ao_cancel ao_run ao_queue fa_close fc_write fc_open ~~fc_request_comp~~	fc_request_comp

The FMSRC class is designed to be used as a component of FMAN which uses it to implement the reading of the source file in the copy file service.

Since the supplied method contains assumptions that the object whose handle is stored in `factive.owner` is an instance of the FMAN class, it is not suitable for use in other situations.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```

CLASS fmsrc fcasy
Source file active object
{
  REPLACE fc_request_comp
}

```

Property

None.

FMSRC methods

FC_REQUEST_COMP

Process a read completion

```
VOID fc_request_comp(VOID);
```

Process the completion of a read from the source file.

The read will always have been of FMAN_READ_SIZE bytes into FMAN's `fman.buf` buffer. On completion of the read `fman.srclen` contains the number of bytes actually read.

If the read completed successfully, it sends an FC_WRITE message to FMAN's FMTARG component to write `fman.srclen` bytes from `fman.buf` to the destination file.

If the read completed with an E_FILE_EOF error, the copy of the file is complete. FMAN's FMTARG component is sent an FA_CLOSE message, the destination file is set to have the attributes and modification date of the source file (an error in reading the attributes of the source file causes `p_leave` to be called). If the copy is of modified files only, the modified (P_FAMOD) attribute of the source file is cleared. An AO_QUEUE message is sent to FMAN's FSCAN component to continue the scan for another file to copy.

If the read completed with any other error, FMAN's FMTARG component is sent an AO_CANCEL message and FMAN is sent an FMAN_ERROR message, which allows the user to skip this file and continue, or abort the copy service.

FMTARG

ACTIVE	FACTIVE	FCASY	FMTARG
q priority isactive pcb stat	owner	recvname flags	
destroy ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	~~fa_close~~ ao_init ~~ao_cancel~~ ao_abrun	ao_cancel ao_run ao_queue fa_close fc_write fc_open ~~fc_request_comp~~	fc_request_comp

The FMTARG class is designed to be used as a component of FMAN, which uses it to implement the writing of the target file in the copy file service.

Since the supplied method contains assumptions that the object whose handle is stored in `factive.owner` is an instance of the FMAN class, it is not suitable for use in other situations.

Class definition

Defined in sub-category file *factive.cl* (generated header file *factive.g*).

```

CLASS  fmtarg  fcasey
Target file active object
{
  REPLACE fc_request_comp
}
```

Property

None.

FMTARG methods

FC_REQUEST_COMP

Process a write completion

```
VOID fc_request_comp(VOID);
```

Process the completion of a write to the target file.

If the write completed successfully, sends FMAN an FMAN_UPDATE message, updates the value of FMAN's `fman.srcLen` to FMAN_READ_SIZE and sends an AO_QUEUE message to FMAN's FMSRC component.

If the write completed with an error, FMAN's FMSRC component is sent an AO_CANCEL message and FMAN is sent an FMAN_ERROR message, which allows the user to skip this file and continue or abandon the copy service.

File manager example

The following example is of a simple application to back up the contents of *M:*, using a console window as the user interface. It formats an SSD in *B:* and copies the entire contents of *M:* to *B:*. The process may be aborted at any time by pressing the ESC key.

The category file, *mcopy.cat*, is as follows:

```

IMAGE mcopy
EXTERNAL olib

INCLUDE appman.g
INCLUDE factive.g
INCLUDE p_keyb.h

CLASS bakfman fman
File manager to perform format and copydev operations
{
    REPLACE fman_complete           Operation now complete
    REPLACE fman_newname            Operation now on this file(s)
    REPLACE fman_update             Update copying display
    REPLACE fman_error              Error in operation, abandon/continue
    PROPERTY
    {
        UWORD count;
        UWORD increment;
        UWORD maxcount;
    }
}

CLASS breakkey active
Looks for Esc key being pressed
{
    REPLACE ao_init
    REPLACE ao_run
    REPLACE ao_queue
    PROPERTY
    {
        P_CON_KBREC k;
    }
}

CLASS bakapp appman
{
    REPLACE am_init
    PROPERTY 2
    {
        PR_BAKFMAN *fman;
        PR_BREAKKEY *breakkey;
    }
}

```

Note that no method function is supplied for the file manager's deferred `fman_fileexist` method. It will never be called in this example since the only file copies are ones that will replace any existing file of the same name.

The code for the application is as follows. Note that pressing Esc during the format will leave the SSD in *B*: unformatted.

```
#include <p_std.h>
#include <p_gen.h>
#include <p_file.h>
#include <p_sys.h>
#include <p_keyb.h>
#include <epoc.h>
#include <mcopy.g>

GLREF_D PR_BAKAPP *w_am;
GLREF_D VOID *winHandle; /* console device handle */

LOCAL_D INT escaped;

#pragma save, METHOD_CALL

METHOD VOID breakkey_ao_init(PR_BREAKKEY *self)
{
    self->active.priority=PRIORITY_ACTIVE_WSERV;
    self->active.pcb=winHandle; /* console device must already be open */
    p_send3(w_am, O_AM_ADD_TASK, self);
    p_send2(self, O_AO_QUEUE);
}

METHOD VOID breakkey_ao_queue(PR_BREAKKEY *self)
/*
Checks for ESC during file manager commands
*/
{
    p_ioc4(self->active.pcb, P_FREAD, &self->active.stat, &self->breakkey.k);
    self->active.isactive=TRUE;
}

METHOD breakkey_ao_run(PR_BREAKKEY *self)
/*
Aborts if ESC pressed
*/
{
    if (self->breakkey.k.keycode!=0x1b)
        p_send2(self, O_AO_QUEUE);
    else
    {
        escaped=TRUE;
        p_printf("\r\nUser aborted");
        p_send2(w_am->bakapp.fman, O_FMAN_CANCEL); /* stops any service cleanly at any
time */
    }
    return(RUN_ACTIVE_USED);
}

METHOD bakfman_fman_error(PR_BAKFMAN *self, INT error)
/*
An error was detected in the operation of the engine.
This method causes the operation to be abandoned and,
in this example, results in the application being
terminated.
*/
{
    escaped=TRUE;
    return(1);
}
```

```

METHOD VOID bakfman_fman_complete(PR_BAKFMAN *self)
/*
The last requested operation is now complete, for ANY reason.
*/
{
    if ((self->fman.action==FMAN_FORMAT)&&(!escaped))
        p_printf("\r\n*** Format successfully completed ***\r\n");
    p_send2(w_am,0_AM_STOP);
}

#define FORMAT_FLAGS (FCOPY_DISP_BLKSIZE|FCOPY_DISP_SFSIZE|FCOPY_DISP_SFNAME)

METHOD VOID bakfman_fman_newname(PR_BAKFMAN *self,FCOPY_DISP *pinfo)
/*
A new file is being acted on by the engine.
*/
{
    if (self->fman.action==FMAN_COPYING)
    {
        if (pinfo->flags&FCOPY_DISP_SFNAME)
            p_printf("Copying %s",pinfo->sfname);
    }
    if (self->fman.action==FMAN_FORMAT)
    {
        /* store some info to avoid accessing it outside this method */
        self->bakfman.count=self->bakfman.maxcount=0;
        if ((pinfo->flags&FORMAT_FLAGS)==FORMAT_FLAGS)
        {
            self->bakfman.increment=pinfo->blksiz;
            self->bakfman.maxcount=pinfo->sfsiz;
            /* copy sfsiz into UWORD since, for format, only lower 16 bits are valid */
            p_printf("Formatting %s",pinfo->sfname);
        }
    }
}

METHOD VOID bakfman_fman_update(PR_BAKFMAN *self)
/*
Update display, as next copy/format block has been processed.
Does not report progress on the copying of files.
*/
{
    if (self->fman.action==FMAN_FORMAT && self->bakfman.maxcount)
    {
        self->bakfman.count+=self->bakfman.increment;
        p_print("\r%05u %05u",self->bakfman.count,self->bakfman.maxcount);
    }
}

METHOD bakapp_am_init(PR_BAKAPP *self)
{
    p_supersend3(self,0_AM_INIT,FLG_APPMAN_CLEAN|FLG_APPMAN_ONLYONE);
    self->bakapp.fman=f_newsend(CAT_MCOPY_MCOPY,C_BAKFMAN,0_FMAN_INIT);
    self->bakapp.breakkey=f_newsend(CAT_MCOPY_MCOPY,C_BREAKKEY,0_AO_INIT);
    return(0);
}

```

```
#pragma ENTER_CALL

LOCAL_C INT RunIt(VOID)
/*
Runs the format and copy services.
Designed to run in an enter harness.
*/
{
    f_leave(p_entsend5(w_am->bakapp.fman,O_FMAN_FORMAT,"LOC:B:\\", "BACKUP",0));
    p_send2(w_am,O_AM_START);
    if (!escaped)
    {
        f_leave(p_entsend5(w_am->bakapp.fman,O_FMAN_COPYDEV,"LOC:M:", "LOC:B:",0));
        p_send2(w_am,O_AM_START);
    }
    p_send2(w_am,O_DESTROY);
    return(FALSE);
}

LOCAL_C INT DoIt(VOID)
/*
Creates and initialises the application manager.
Runs the format and copy services under a further enter harness.
Designed to run in an enter harness to catch initialisation failures.
*/
{
    INT err;

    escaped=FALSE;
    w_am=(PR_BAKAPP *)f_new(CAT_MCOPY_MCOPY,C_BAKAPP);
    p_send2(w_am,O_AM_INIT);
    err=p_enter1(RunIt);
    p_send2(w_am,O_DESTROY);
    return(err);
}

#pragma restore

GLDEF_C main(VOID)
/*
Copy all of M: to an SSD in B:
*/
{
    p_linklib(0);
    p_printf("Backing up M: to B:"); /* convenient way to start up the console device
*/
    return(p_enter1(DoIt));
}
```

The nested `p_enter` protection for the calls to `DoIt` and `RunIt` ensure that the application manager is not sent a `DESTROY` message if it fails to be created, but is destroyed in the event of any other failure.

The `DESTROY` message is not strictly necessary since the operating system will clean up all resources used by an application when the application terminates. Nevertheless, it is good practice to ensure that an application is in a fit state to free its resources at any time.

In this case, by use of the auto-destruction mechanism, destroying the application manager will cause the destruction of both the file manager and the `BREAKKEY` active object. The superclass active object destroy method ensures that any outstanding console keyboard read is cancelled and that the console device (whose handle is stored in `active.pcb`) is closed.

CHAPTER 17

THE LOCS LOCAL FILE SCAN CLASS

LOCS
flags pcb pname match info name wildname
ls_matchname ls_scan
<i>ls_filename</i>

The LOCS abstract class provides a set of methods to perform a synchronous scan of the *LOC::* and *ROM::* filing systems to locate file names which match a wildcarded name. It is similar in purpose to, but simpler than, the asynchronous scanning classes described in the *File Active Objects* and *File Lists* chapters.

LOCS uses a synchronous scan, so once a scan has been started no other events can be processed until the scan completes. However, since file system requests on the *LOC::* and *ROM::* filing systems complete very quickly, this is unlikely to cause any significant loss of responsiveness to user input.

LOCS must be subclassed to provide the deferred LS_FILENAME method, to process matching file names.

Precursors

The reader is assumed to understand:

- the PLIB/EPOC file system directory read functions
- the *LOC::* and *ROM::* filing systems

Class definition

The `LOCS` class subclasses `ROOT` and is defined in the sub-category file *factive.cl* (with generated header file *factive.g*).

```
CLASS locs root
Synchronous Local/ROM filing system file finder
{
  ADD ls_matchname      Matching name checker
  ADD ls_scan           Start the scan off
  DEFER ls_filename     Process located file name
  CONSTANTS
  {
    LOCS_FLG_ROM        0x1000
    LOCS_FLG_LOC        0x2000
    LOCS_FLG_ROOT       (0x4000|LOCS_FLG_LOC)
    LOCS_FLG_ROOT_ONLY  0x4000          Internal to locs only
  }
  PROPERTY
  {
    UWORD flags;          Controlling flags
    UBYTE *pcb;           Open directory file handle
    UBYTE *pname;         Where to read names into
    UBYTE *match;         Match name string
    P_INFO info;          Currently found file info
    UBYTE name[P_FNAME_SIZE];
    UBYTE wildname[P_FNAME_SIZE];
  }
}
```

Property

<code>locs.flags</code>	controlling flags to drive the scan. This field should not be accessed by any subclass.
<code>locs.pcb</code>	the currently opened directory file handle. It should not be accessed by any subclass.
<code>locs.pname</code>	a pointer to the name and extension within the full file specification in the <code>locs.name</code> buffer. A subclass may use this pointer to read the file name.
<code>locs.match</code>	a pointer to the name and extension within the wildcard full file specification in the <code>locs.wildname</code> buffer. Used to determine if a generated file name matches the name requirements. A subclass should treat this as read only.
<code>locs.info</code>	the file information corresponding to the file name in <code>locs.name</code> , read from the currently opened directory file. A subclass should treat this as read only.
<code>locs.name</code>	the full file specification of the current file, whose file information is in <code>locs.info</code> . A subclass should treat this as read only.
<code>locs.wildname</code>	contains a parsed version of the wildcarded name passed to the <code>ls_scan</code> method. If necessary, it is modified during the scan in order to search the <i>ROM::</i> and/or the root directories of the <i>LOC::</i> filing systems.

LOCS methods

J LS_SCAN

Perform a file scan

```
VOID ls_scan(UBYTE *pname, INT flags);
```

Perform a synchronous search for files with names matching the wildcard string pointed to by `pname` and of type and location specified by `flags`. This method will not return until the scan is complete.

The value of `flags` may be any ORED combination of items from the following two groups:

P_FAMOD	select only modified files
P_FAHIDDEN	include hidden files
P_FASYSTEM	include system files
LOCS_FLG_ROM	scan the ROM:: device (scanned first)
LOCS_FLG_LOC	scan the specified directory of all LOC:: devices (in alphabetical order)
LOCS_FLG_ROOT	scan the root directory of a LOC:: device before scanning any specified directory

The method first copies the value of `flags` into `locs.flags` and parses the string pointed to by `pname` into the `locs.wildname` buffer. It then reads file names, from the directory files specified by `pname` and `flags`, into the `locs.name` buffer. For each file name it sends an LS_MATCHNAME message. If this returns TRUE it also sends an LS_FILENAME message. The sending of this message is under the protection of `p_enter` so that the `ls_scan` method itself will never be terminated by a call to `p_leave`.

The scan is terminated either when there are no more file names to read or when the `ls_filename` method returns a non-zero value.

For example:

```
p_send4(locs, 0_LS_SCAN, "fon\\*.fon", LOCS_FLG_ROM|LOCS_FLG_ROOT|LOCS_FLG_LOC);
```

will find all *.fon* files in the ROM, and in the root and *\fon* directories of all SSDs in the LOC:: filing system.

J LS_MATCHNAME

Check file name match

```
INT ls_matchname(VOID);
```

Check the file name pointed to by `locs.pname`, whose file type information is in `locs.info`, against the wildcarded name pointed to by `locs.match` and the file types in `locs.flags`.

This method is not called when the name is of a volume or a directory file.

Returns TRUE if the file matches the specified requirements else FALSE.

Deferred LOCS methods

LS_FILENAME

Process a file name

```
INT ls_filename(UBYTE *pname);
```

This method should contain the logic to process the name of a file that matches the initial specifications as passed to the `ls_scan` method.

It is called from the `ls_scan` method each time the `ls_matchname` method returns TRUE, with `pname` pointing to the full file specification in the `locs.name` buffer. If the subclass is only interested in the file name and extension, it may read this via `locs.pname`.

The method is expected to return FALSE to continue the scan or, having found a file that satisfies its requirements, it may return TRUE to terminate the scan immediately, causing the `ls_scan` method to return.

If this method does not return TRUE the scan will terminate when the LS_SCAN method has no more file names to read.

CHAPTER 18

SYSTEM SERVICES

Each member of the SIBO family of machines supplies one or more of the following global system services:

- to allocate or replace an icon position, or to remove an icon
- to run a file-based application by specifying the file that it is to open
- to nominate the current link paste server
- to provide the process id of the current link paste server, if any

All these services are available on MC 200/400 machines, where they are provided by the System process, SYS\$SHLL.

On Series 3 and HC machines, only the two link paste related services are available and are supplied by the window server process, SYS\$WSRV.

The available services are accessed by means of an inter-process message being sent to the supplying process. The SYSTEM class provides a simplified form of access that hides the inter-process messaging mechanism.

Precursors

The reader is assumed to understand:

- the initialisation of the APPMAN class

SYSTEM

SYSTEM
pid
sy_init sy_icon_pos sy_link_server sy_link_paste sy_exec_open

The SYSTEM class provides simple access to the available system services, as described above.

An application does not normally explicitly create and initialise an instance of the SYSTEM class. On MC and Series 3 machines, an instance of the SYSTEM class is usually created and initialised automatically by passing a flag (FLG_APPMAN_SYSTEM on the MC, or FLG_APPMAN_LINKING on the Series 3) to the application manager's AM_INIT method. In this case the handle of the created and initialised SYSTEM instance is stored in the application manager's appman.system property field.

Note that FLG_APPMAN_LINKING (which is defined in *hwimman.g*) and FLG_APPMAN_SYSTEM cause the appropriate process name (SYS\$SHLL or SYS\$WSRV respectively) to be used by the application manager's am_init method.

Class definition

The SYSTEM class subclasses ROOT and is defined in the sub-category file *appman.cl* (with generated header file *appman.g*).

```
CLASS system root
  For access to system services
  {
    ADD sy_init          Gets the pid of the supplying process
    ADD sy_icon_pos      Allocate/remove/replace an icon position
    ADD sy_link_server   Nominate oneself as the link paste server
    ADD sy_link_paste    Get pid and format mask of link paste server
    ADD sy_exec_open     Execute an application to open specified file
    CONSTANTS
    {
      IC_SYSTEM_ALLOC    0    Allocate an icon position
      IC_SYSTEM_REMOVE   1    Remove an icon position
      IC_SYSTEM_REPLACE  2    Replace an icon position
    }
    PROPERTY
    {
      UWORD pid;          Process id of shell
    }
  }
}
```

Property

system.pid the process id of the System process. This is private to the SYSTEM class and should not be accessed by any other code

SYSTEM methods

J SY_INIT

Initialise

```
VOID sy_init(TEXT *sysnam);
```

Initialise, by finding - and storing in system.pid - the process id of the System application whose application name is sysnam.

Returns zero if successful. On error, writes nothing to system.pid and returns the relevant error number.

As explained earlier, this method is not normally called explicitly by application code. It is called by the application manager in its am_init method which supplies the appropriate text.

SY_ICON_POS

Control icon positioning

```
VOID sy_icon_pos(UINT req, P_POINT *pos, UINT prev);
```

This service is only available on MC 200/400 machines.

Control the positioning of icons representing tasks. The behaviour depends on the value of req, as follows:

IC_SYSTEM_ALLOC	Allocate an icon position, writing the allocated position in *pos and returning an index corresponding to this position. If prev is -1, the icon is allocated at the first free position. Otherwise prev should be an index returned by a previous SY_ICON_POS message, when an attempt will be made to allocate the corresponding position. If this position is occupied, the first free position is allocated, as for a prev of -1.
IC_SYSTEM_REMOVE	Remove the icon at position *pos, which should contain a position previously generated by an SY_SYSTEM_POS message with a req of either IC_SYSTEM_ALLOC or IC_SYSTEM_REPLACE. The value of prev is ignored. Returns zero.
IC_SYSTEM_REPLACE	Replace the position corresponding to the index value in prev (which should be an index returned by a previous SY_ICON_POS message) with the position in *pos. The value in *pos is updated to be either the 'snap' position nearest to the passed position, or the original position if this nearest position is occupied. Returns an index corresponding to the new position.

SY_LINK_SERVER**Set link paste server**

```
INT sy_link_server(ULONG fmask);
```

Nominate the current process (that is, the process which sends this message) as the link paste server.

The value of `fmask` is a bit mask of the formats in which the server is prepared to provide data. The formats are discussed in the description of the `LINKSV` class definition in the *Link Paste* chapter of this manual.

Returns zero.

SY_LINK_PASTE**Get link paste server**

```
INT sy_link_paste(ULONG *pfmt);
```

Find the process id of the application (if any) that is the current link paste server.

If such a process exists, writes the bit mask of available formats to `*pfmt` and returns the process id. The formats are discussed in the description of the `LINKSV` class definition in the *Link Paste* chapter of this manual.

If there is no current link paste server, writes nothing to `*pfmt` and returns zero.

SY_EXEC_OPEN**Run an application by file name**

```
INT sy_exec_open(TEXT *pname);
```

This service is only available on MC 200/400 machines.

Queue a request to run the appropriate application that will open the file whose name is pointed to by `pname`.

The application is selected on the basis of the file name extension together with the application/extension associations declared in one or more *def.ext* files.

Returns zero without waiting for the request to complete.

CHAPTER 19

INTER-PROCESS COMMUNICATION

This chapter describes the `IPCS` and `SERVER` classes that may be used to receive and process inter-process messages from one or more sources.

A process running under EPOC may open only one message channel for receiving inter-process messages. Inter-process messages may, however, arrive from a variety of sources. In order to distinguish between messages from different sources, it is conventional to group them, assigning a range of message type values to each source. (The type value is stored in the `type` field of the `E_MESSAGE` struct that forms the header of each inter-process message.)

The following message groups are defined and used by existing software:

CONSOL	message types 0 to 0x0f (MC 200/400 only)
TOPLIP	message types 0x10 to 0x1f (MC 200/400 only)
LINKSV	message types 0x20 to 0x2f (link paste)
Automatic test system	message types 0x30 to 0x3f

It is usually convenient to use a separate *server object* (by definition, a subclass of `SERVER`) to process the receipt of inter-process messages of each group. For example, all message types concerned with link paste should be handled by a link paste server object.

The `IPCS` class provides the central mechanism by which a process receives an inter-process message and directs it to the appropriate server object. The relationship between `IPCS` and server objects is analogous to that between `APPMAN` and active objects.

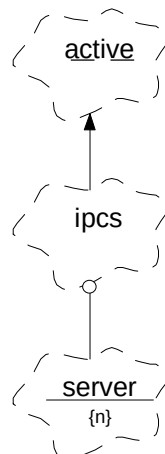
It may be noted that a process only needs to create an instance of the `IPCS` class (a message channel) if it receives messages; a process may send an inter-process message without opening a message channel.

Precursors

The reader is assumed to understand:

- the `ACTIVE` class and the application manager's event scheduling mechanisms
- the `PLIB` inter-process messaging services, described in the *Processes and Inter-process Messaging* chapter of the *PLIB Reference* manual
- the `p_enter` and `p_leave` error handling services

Inheritance tree



IPCS

ACTIVE	IPCS
q priority isactive pcb stat	hd
~~destroy~~ ~~ao_init~~ ~~ao_cancel~~ ~~ao_abrun~~ ~~ao_queue~~ ~~ao_run~~	destroy ao_init ao_cancel ao_queue ao_run ao_abrun ip_add_server

The IPCS class is an active object that provides services to receive inter-process messages and dispatch them to the appropriate server object.

This class should be used when several servers are required, to process messages of more than one group. It should also be used if the application is to support link paste. The application can then take advantage of the link paste server (described in the following chapter) even if no other servers are required.

In other cases, where only one group of message types is to be processed, it may be more convenient to subclass IPCS so that the messages are processed in the subclass `ao_run` method, rather than in a separate server.

An instance of the IPCS class is created and initialised automatically during the application manager's `am_init` method, provided the `FLG_APPMAN_IPCS` flag is set, and its handle is written to `appman.ipcs`.

If you subclass IPCS you must explicitly create and initialise the instance yourself. You must not specify the `FLG_APPMAN_IPCS` flag in the `AM_INIT` message to APPMAN (attempting to open more than one messaging channel will cause the application to fail). You may, however, store the handle in the application manager's `appman.ipcs` so that the object will be destroyed automatically when the application manager is destroyed at termination of the application.

Class definition

Defined in sub-category file *appman.cl* (generated header file *appman.g*).

```

CLASS ipcs active
The ipc server active object for inter-process communication
{
  REPLACE destroy          Set pcb to NULL as not really handle based
  REPLACE ao_init          Init message queue, add to appman queue, read
  REPLACE ao_cancel        Cancel a queued message read
  REPLACE ao_queue         Queue a message read
  REPLACE ao_run           Despatch message to a server
  REPLACE ao_abrun         Queue a read and supersend
  ADD ip_add_server        Add a server to the queue
  PROPERTY
  {
    P_QUE hd;              Server queue header
  }
}

```

Property

<code>ipcs.hd</code>	the head of the queue of server objects that can accept IPC messages. It should not be read or manipulated by any subclass.
----------------------	---

IPCS methods

J DESTROY

Destroy

```
VOID destroy(VOID);
```

Set `active.pcb` to `NULL` and supersend the `DESTROY` message.

Any non-zero value in `active.pcb` is a pointer to a message buffer for a received message; this is set in the `ao_queue` method. If `active.pcb` is non-zero, the superclass `destroy` method assumes that it is an I/O channel and attempts to close it. Setting it to zero avoids the problem (and no information is "lost" by doing so).

AO_INIT

Initialise

```
VOID ao_init(UINT size, UINT num);
```

Initialise the IPCS object by performing a number of activities.

- Initialise an empty server queue in `ipcs.hd`
- Initialise a message queue of `num` messages; each message slot will consist of an `E_MESSAGE` header *plus* a buffer of length `size`. The queue is initialised using the PLIB function `p_init`. Note that it is the receiver that specifies the size (and, by implication, the structure) of an inter-process message.
- If the creation of the message queue is successful, the object adds itself, with priority `PRIORITY_ACTIVE_IPCS`, to the application manager's active object task queue and then sends itself an `AO_QUEUE` message.

Calls `p_leave` on error.

AO_QUEUE

Queue a message read

```
VOID ao_queue(VOID);
```

If a message read is currently outstanding, indicated by `active.isactive` *not* set to `FALSE`, the method does nothing.

If a message read is not outstanding, it queues a message read by calling `p_mreceive`, using `active.stat` as the completion status word. On completion of the read, the pointer to the received message slot will have been written to `active.pcb`.

If the call to `p_mreceive` succeeds, `active.isactive` is set to `TRUE`. The call to `p_mreceive` will panic if messages have not been initialised or if an asynchronous message receive request is already pending.

J AO_CANCEL

Cancel read request

```
VOID ao_cancel(VOID);
```

If a message read is *not* currently outstanding, indicated by `active.isactive` set to `FALSE`, the method does nothing.

If a message read is outstanding, it calls `p_mcancel` to cancel any pending asynchronous request to receive a message and then waits (with `p_waitstat` on `active.stat`) for the cancel to complete; it then sets `active.isactive` to `FALSE`.

AO_RUN

Process a message

```
INT ao_run(VOID);
```

Pass the message to one of the servers in the server queue.

Scans the items in the server queue until one is found that is prepared to process the type of message that has arrived (by comparing the message type with the upper and lower message type limits stored in each server's property). Calls `p_panic(P_PANIC_P_IPCS_2)` if no such server is found.

The first server that is prepared to process the message is sent an `SV_RUN` message under the protection of a `p_enter` (any error in the server's `sv_run` method is expected to result in a call to `p_leave` with a non zero error number).

On detection of such an error the server is sent an `SV_ABRUN` message, after which the `ao_run` method itself calls `p_leave` to propagate the error (thus causing the receipt of an `AO_ABRUN` message).

If there are no errors, the `ao_run` method sends an `AO_QUEUE` message to read another message and then returns `RUN_ACTIVE_USED`.

AO_ABRUN **Handle error**

```
VOID ao_abrun(VOID);
```

Send an `AO_QUEUE` message to read another message before supersending the `AO_ABRUN` message.

IP_ADD_SERVER **Add item to server queue**

```
VOID ip_add_server(PR_SERVER *hand);
```

Add the server object pointed to by `hand` to the end of the server queue which is anchored in `ipcs.hd` (i.e. in the `IPCS` object).

Unlike the application manager's queue, there is no priority system. If two servers can both handle a message of a particular type, the one that was first added to the server queue will be sent the `SV_RUN` message.

Such a state is, in general, a programming error. Normally, each server is expected to handle a unique range of message types. Note that the message types handled by a server fall into a single range.

SERVER

SERVER
q t1 t2 cid
destroy sv_abrun sv_init <i>sv_run</i>

The `SERVER` abstract class defines a set of services for processing a received inter-process message.

`SERVER` must be subclassed to provide at least an `sv_run` method in order to create a useful server object. An example, the link paste server object, is described in the following chapter.

Server objects work closely with an instance of the `IPCS` class (described above) whose handle is stored in the application manager's `appman.ipcs`.

Class definition

Defined in sub-category file *ipc.cl* (generated header file *ipc.g*).

```

CLASS server root
{
  REPLACE destroy          Dequeue and supersend
  ADD sv_abrun=p_dummy     Handle leave from sv_run
  ADD sv_init              Queue to ipcs
  DEFER sv_run             Process message
  PROPERTY
  {
    P_QUE q;               Queue header
    UWORD t1;              Server type range (lowest msg number)
    UWORD t2;              (highest msg number)
    UWORD cid;             Client process id, or zero for any process
  }
}

```

Property

<code>server.q</code>	used to add the server to an IPCS server queue. It should not be accessed by any subclass.
<code>server.t1</code>	the lowest message type that is acceptable to this server. It is read by IPCS and should not otherwise be accessed.
<code>server.t2</code>	the highest message type that is acceptable to this server. It is read by IPCS and should not otherwise be accessed.
<code>server.cid</code>	intended for the storage of a client process id, if any. It is not used by either the <code>SERVER</code> or <code>IPCS</code> classes and is therefore free for use by any subclass. It may be used to store other information, if so required.

SERVER methods

J DESTROY

Destroy

```
VOID destroy(VOID);
```

If the server object sits in an IPCS server queue, i.e the queue anchored in `ipcs.hd`, remove it from the queue using `p_deque`. Then supersend a `DESTROY` message.

SV_INIT

Initialise

```
VOID sv_init(UINT t1, UINT t2);
```

Send an `IP_ADD_SERVER` message to the object whose handle is stored in the applications manager's (`w_am`) `appman.ipcs` property field. This object is assumed to be an instance of (a subclass of) the `IPCS` class.

Copies `t1` and `t2` to `server.t1` and `server.t2` respectively, to set the range of message types acceptable to this server. It is assumed that `t2` is greater than (or equal) to `t1`.

SV_ABRUN

Handle error

```
VOID sv_abrun(VOID);
```

This method does nothing. It is called by IPCS, following a `p_leave` in a server's `sv_run` method.

A subclass may replace this method to perform specific error handling, over and above that subsequently executed in the `IPCS` `ao_abrun` method.

This method is not expected to call `p_leave`.

Deferred SERVER methods

SV_RUN

Process a message

```
VOID sv_run(E_MESSAGE *pmess);
```

Process the message pointed to by pmess.

This message is sent by IPCS when it has determined that the message that has arrived is within the range of types that this server is prepared to process.

CHAPTER 20

LINK PASTE

Link paste is the term used for the transfer of data from one application to another, for example to transfer a database record into the document currently being edited in a word processor. Link paste is initiated by the Bring command in a Series 3 application and by a Link command in an MC 200/400 application.

To clarify terminology, this chapter refers to a *receiver* and a *supplier*. A receiver is the process which asks for the data while the supplier is the process which provides that data. In the context of link-paste, a *client* is synonymous with a receiver while a *server* is synonymous with a supplier.

The following description applies to applications written for the Series 3 or the MC 200/400. It does not apply to applications on the HC, since in this case the `SYSTEM` class does not automatically provide access to the appropriate system services (see the *System Services* chapter of this manual). In general, custom applications written for the HC will need to provide their own means (usually by explicit inter-process messaging) for the receiver to identify a suitable supplier. Once this is done, however, the data may be transferred using instances of `LINKCL` and a subclass of `LINKSV`, as described below.

Link paste is implemented by means of inter-process messages sent by the receiver which are handled by the supplier of the data. The supplier uses a link paste server object which is generally an instance of a subclass of `LINKSV` (link server). The receiver generally uses an instance of the `LINKCL` (link client) class.

A supplier of data is the passive partner in the transfer in the sense that it replies to IPCS messages sent by the receiver. A supplier does *not* send IPCS messages to the receiver. (See the description of `p_msendreceivev` in the *PLIB Reference Manual*)

The supplying process must, however, register itself if it is capable of supplying data and must be able to specify in which set of data formats it is prepared to supply the data. A Series 3 or MC 200/400 application may register itself as the current supplier by sending an `SY_LINK_SERVER` message to an owned instance of the `SYSTEM` class. For example, if a word processor has a highlighted region of text when it is sent into background, it will, in general, register itself as the current supplier at that point.

The range of possible formats are declared, for convenience, in the `LINKSV` class definition. Their interpretation is significant to the application rather than to the `LINKCL` and `LINKSV` classes.

A typical transaction is initiated by the receiver and requires the receiver to perform the following activities:-

- It must first locate a process that is prepared to provide data in a suitable format. In the case of an application running on either the Series 3 or an MC machine, this is done by sending an `SY_LINK_PASTE` message to an owned instance of the `SYSTEM` class to retrieve both the process id of the process which is currently registered as a supplier of data and a bit mask of the available data formats.
- It must, at some stage, create an instance of the `LINKCL` class and send it an `LC_START` message, passing the process id of the supplier and the particular format, from those available, in which the data is required.
- It sends one or more `LC_GET_DATA` messages (to the `LINKCL` object), passing both the address of a buffer which is ready to receive the data and the maximum length of data which can be handled. Once data has been received, it can be transferred into the application's own data structures.
- This should continue until either the available data is exhausted, in which case the transaction is automatically terminated, or until the receiver does not wish to receive further data. In this second case the receiver must send an `LC_STOP` message (to the `LINKCL` object) to terminate the transaction.

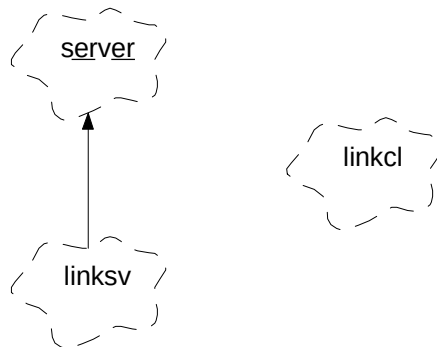
The methods in the `LINKCL` and `LINKSV` classes and the particular values that their parameters take, in a sense, establish a protocol for communication between the receiver and supplier.

Precursors

The reader is assumed to understand:

- inter-process messaging
- the `SERVER` and `IPCS` classes

Class diagram



LINKCL

LINKCL
pid
destroy lc_start lc_get_data lc_stop

The `LINKCL` class provides the methods by which a process may initiate a link paste data transfer, receive one or more sections of data and, if necessary, terminate the transaction.

Class definition

Defined in sub-category file *ipc.cl* (generated header file *ipc.g*).

```
CLASS linkcl root
The link client
{
  REPLACE destroy      Stop the transaction then supersend
  ADD lc_start          Start a transaction
  ADD lc_get_data       Get a data record
  ADD lc_stop           Stop the transaction
  PROPERTY
  {
    UWORD pid;          Process ID of link server
  }
}
```

Property

<code>linkcl.pid</code>	the process id of the process that is currently acting as the link paste server
-------------------------	---

LINKCL methods

J DESTROY

Destroy

```
VOID destroy(VOID);
```

Stops any currently outstanding transaction with the link paste server by sending an LC_STOP message before supersending the DESTROY message.

LC_START

Initiate a transaction

```
VOID lc_start(INT pid, INT format);
```

Initiate a link paste transaction, requesting data of the type specified by `format` from the current link paste server process.

`pid` is the process id of the current link paste server as retrieved by sending a SY_LINK_PASTE message to the receiver's owned SYSTEM object.

Records the link paste server's process id by setting `linkcl.pid` to `pid` and then sends a TY_LINKSV_STEP inter-process message, containing the required data format, to the current link paste server process.

Calls `p_leave` on error.

LC_GET_DATA

Request data

```
UINT lc_get_data(UBYTE *buf, UINT len);
```

Request up to `len` bytes of data to be written to the buffer pointed to by `buf`.

Sends a TY_LINKSV_STEP inter-process message, containing the buffer pointer and maximum length, to the link paste server.

If there is no more data to receive, `linkcl.pid` is set to zero, automatically terminating the transaction. A further LC_START message is required in order to receive the data again.

Returns one of:

- the (positive) length of data written by the link paste server to the buffer,
- E_FILE_EOF if there is no more data to receive,
- E_GEN_FAIL if the transaction has not been initiated.

Calls `p_leave` on all other errors.

LC_STOP

Terminate a transaction

```
VOID lc_stop(VOID);
```

Terminate any transaction with a link paste server.

It effectively cancels any current transaction and then sets `linkcl.pid` to zero. It is harmless if there is no current transaction.

An LC_START message is required to start a further transaction.

Calls `p_leave` on error.

LINKSV

SERVER	LINKSV
q t1 t2 cid	buf len
destroy sv_abrun ~~sv_init~~ ~~sv_run~~	sv_init sv_run ls_set_format ls_get_data

The LINKSV abstract class provides the basic mechanisms for supplying, on request, sections of application data in one of a variable number of formats.

LINKSV must be subclassed to supply the two deferred methods which depend on the way that the application interprets the data formats. The formats are discussed in the *HWIM Reference* manual.

Class definition

Defined in sub-category file *ipc.cl* (generated header file *ipc.g*).

```

CLASS linksv server
The link paste server
{
  REPLACE sv_init          Supersend with parameters
  REPLACE sv_run           Process message
  DEFER ls_set_format      Set the desired format
  DEFER ls_get_data        Get a data record
  CONSTANTS
  {
    ! Message types
    TY_LINKSV_STEP         0x21      Link paste step
    TY_LINKSV_DEATH        0x22      Death of client
    ! Data formats - formats 0 to 31 inclusive are reserved for use by Psion
    DF_LINK_NATIVE         0         known only to another invocation of the
supplying application
    DF_LINK_TEXT           1         plain ASCII text
    DF_LINK_TABTEXT        2         ASCII text including tab characters
    DF_LINK_VOICE          3         voice processor data
    DF_LINK_PARAS          4         text with a paragraph structure
    DF_LINK_SPR            5         spreadsheet data
    DF_LINK_WRD            6         word processor data (only for Series 3a and later
machines)
    DF_LINK_AGD            7         agenda data (only for Series 3a and later machines)
  }

  PROPERTY
  {
    UBYTE *buf;            Set by li_get_data
    UWORD len;             Set by li_get_data
  }
}

```

Property

linksv.buf	a pointer to data, in the required format, available for copying to the client. This is read by the sv_run method and should be set by the deferred ls_get_data method.
linksv.len	the length of the data pointed to by linksv.buf. This is read by the sv_run method and should be set by the deferred ls_get_data method.

LINKSV methods

SV_INIT

Initialise

```
VOID sv_init(VOID);
```

Supersend the SV_INIT message, specifying TY_LINKSV_STEP and TY_LINKSV_DEATH as the lower and upper inter-process message types that this server is prepared to handle.

SV_RUN

Process a message

```
VOID sv_run(VOID);
```

Process a received link paste inter-process message in the range TY_LINKSV_STEP to TY_LINKSV_DEATH.

The handling of an inter-process message is fairly complex and depends on a number of factors, not least of which is the type of inter-process message received!

Inter-Process Message TY_LINKSV_STEP

An inter-process message of type TY_LINKSV_STEP is part of a sequence of such messages transferring data from the client to the server. The method distinguishes between the first TY_LINKSV_STEP inter-process message and subsequent inter-process messages of this type.

First TY_LINKSV_STEP inter-process message

The first TY_LINKSV_STEP inter-process message represents a new transaction and corresponds to the client sending an LC_START message to an instance of its LINKCL object. The link-server object decides that this is the first time if the property server.cid is zero.

The following is done:-

- The process ID of the client is saved in server.cid.
- p_logon is called to request that the SYS\$MANG process send it a TY_LINKSV_DEATH inter-process message if the client terminates.
- The TY_LINKSV_STEP inter-process message contains the required data format and this is passed as the parameter to an LS_SET_FORMAT message which should register the format in which the data is to be provided.
- The inter-process message is freed by calling p_mfree with a reply value of zero.

Subsequent TY_LINKSV_STEP inter-process messages

Subsequent TY_LINKSV_STEP inter-process messages are expected to contain a pointer to a buffer and a maximum length in the message data, as set by the LINKCL lc_get_data method. Two situations must be handled - the buffer pointer is zero or non-zero.

Zero buffer pointer

A zero buffer pointer is taken to mean that the client wishes to terminate the link paste transaction. This is effectively a cancel and corresponds to the client sending an LC_STOP message to an instance of its LINKCL object.

The following is done:-

- an LS_GET_DATA message is sent with a parameter of -1. The effect of this is to cause the deferred ls_get_data method to tidy and reset the link server object in preparation for a new transaction.
- p_logoff is called to cancel the request that the SYS\$MANG process send a TY_LINKSV_DEATH inter-process message on termination of the client.
- server.cid which contains the process id of the link paste client (i.e. the receiver), is set to zero, therefore losing all knowledge of that client
- The inter-process message is freed (p_mfree) returning E_FILE_EOF to the client.

Non-zero buffer pointer

A non-zero buffer is taken to mean that the client is making a request for (more) data. An `LS_GET_DATA` message is sent to retrieve an amount of data not exceeding the maximum length.

If the `ls_get_data` method returns a *non-zero* value:-

- it is assumed that `linksv.buf` and `linksv.len` have been set to indicate available data, which is copied to the client's buffer.
- The inter-process message is freed (`p_mfree`) returning a value equal to the length of the data copied to the client's buffer.

If the `ls_get_data` method returns a *zero* value:-

- this is taken to mean that no more data is available.
- `p_logoff` is called to cancel the request that the `SYSDMANG` process send a `TY_LINKSV_DEATH` inter-process message on termination of the client.
- `server.cid` which contains the process id of the link paste client is set to zero, therefore losing all knowledge of that client
- The inter-process message is freed (`p_mfree`) returning `E_FILE_EOF` to the client.

Inter-Process Message `TY_LINKSV_DEATH`

An inter-process message of type `TY_LINKSV_DEATH` means that the link paste client died or was terminated during the inter-process data transfer.

On receipt of this inter-process message, the following is done:-

- The inter-process message is freed (`p_mfree`)
- `server.cid` which contains the process id of the link paste client is set to zero, therefore losing all knowledge of that client
- an `LS_GET_DATA` message is sent with a parameter of -1. The effect of this is to cause the deferred `ls_get_data` method to tidy and reset the link server object in preparation for a new transaction.

Deferred LINKSV methods

LS_SET_FORMAT

Set data format

```
VOID ls_set_format(UINT format);
```

Register the data format in which data is to be provided. For link paste initiated by one of the built-in applications, the format will be one of the `DF_LINK_XXX` values listed in the `LINKSV` class definition.

The method should call `p_leave` on error.

LS_GET_DATA

Provide data

```
INT ls_get_data(UINT len);
```

Provide not more than `len` bytes of data in the currently specified format.

A pointer to the data and its length should be written to `linksv.buf` and `linksv.len` respectively. The method should return zero if there is no more data available, else a positive number.

A value of -1 for `len` indicates that the client process no longer requires data. In this case the method should tidy up and reset any variables so that future `LS_GET_DATA` messages read the data from the start of available data.

The method should call `p_leave` on error.

INDEX

- .trm files
 - serial port parameters, 8-11
- ACTIVE class
 - AO_ABRUN method, 11-4
 - AO_CANCEL method, 11-3
 - AO_INIT method, 11-3
 - AO_QUEUE method, 11-3
 - AO_RUN method, 11-4
 - DESTROY method, 11-3
 - methods, 11-3
 - oop, 11-1
- active objects
 - asynchronous I/O devices, 11-1
 - class, 11-1
 - file, 14-1
 - file server and, 11-1
 - priorities, 10-2
 - return value, 10-3
 - scheduling mechanism, 10-6
 - scheduling, 10-3
 - TIMER, 13-1
- AIDLE class
 - AO_INIT method, 12-2
 - AO_RUN method, 12-2
 - compute intensive tasks, 12-1
 - example, 12-3
 - idle time computation, 12-3
 - methods, 12-2
 - oop, 12-1
 - pause an operation, 12-3
- AIDLE example
 - oop, 12-3
- AM_ADD_TASK
 - APPMAN class method, 10-9
- AM_CHANGE_PRI
 - APPMAN class method, 10-12
- AM_CLEAN_UP
 - APPMAN class method, 10-11
- AM_FINDIMG
 - APPMAN class method, 10-11
- AM_INIT
 - APPMAN class method, 10-5
- AM_LOAD_RES_BUF
 - APPMAN class method, 10-10
- AM_LOAD_RESOURCE
 - APPMAN class method, 10-9
- AM_NOTIFY
 - APPMAN class method, 10-10
- AM_NOTIFYERR
 - APPMAN class method, 10-11
- AM_ONLYONE
 - APPMAN class method, 10-12
- AM_RSCNAME
 - APPMAN class method, 10-10
- AM_START
 - APPMAN class method, 10-6
- AM_STOP
 - APPMAN class method, 10-9
- AM_WAIT
 - APPMAN class method, 10-6
- ANIMATOR class
 - AO_INIT method, 13-4
 - AO_RUN method, 13-4
 - methods, 13-4
 - oop, 13-3
- AO_ABRUN
 - ACTIVE class method, 11-4
 - FACTIVE class method, 14-3
 - FMFMT class method, 16-14
 - FMSCAN class method, 16-15
 - IPCS class method, 19-4
 - PNODE class method, 15-4
 - PSEL class method, 15-7
- AO_CANCEL
 - ACTIVE class method, 11-3
 - BUZSND class method, 13-6
 - FACTIVE class method, 14-2
 - FCASY class method, 14-12
 - IPCS class method, 19-3
 - PSEL class method, 15-7
- AO_INIT
 - ACTIVE class method, 11-3
 - AIDLE class method, 12-2
 - ANIMATOR class method, 13-4
 - BUZSND class method, 13-5
 - FACTIVE class method, 14-2
 - IPCS class method, 19-3
 - PSEL class method, 15-7
 - TIMER class method, 13-2
- AO_QUEUE
 - ACTIVE class method, 11-3
 - BUZSND class method, 13-6
 - FCASY class method, 14-12
 - FCSYNC class method, 14-15
 - FNODE class method, 14-9
 - FSCAN class method, 14-5
 - IPCS class method, 19-3
 - TIMER class method, 13-2
- AO_RUN
 - ACTIVE class method, 11-4
 - AIDLE class method, 12-2
 - ANIMATOR class method, 13-4
 - BUZSND class method, 13-6
 - FCASY class method, 14-13
 - FMFMT class method, 16-13
 - FMMK class method, 16-12
 - FNODE class method, 14-9
 - FSCAN class method, 14-5
 - IPCS class method, 19-3
- APPMAN class
 - AM_ADD_TASK method, 10-9
 - AM_CHANGE_PRI method, 10-12
 - AM_CLEAN_UP method, 10-11
 - AM_FINDIMG method, 10-11

- AM_INIT method, 10-5
- AM_LOAD_RES_BUF method, 10-10
- AM_LOAD_RESOURCE method, 10-9
- AM_NOTIFY method, 10-10
- AM_NOTIFYERR method, 10-11
- AM_ONLYONE method, 10-12
- AM_RSCNAME method, 10-10
- AM_START method, 10-6
- AM_STOP method, 10-9
- AM_WAIT, 10-6
 - definition, 10-4
 - diagram, 10-4
 - methods, 10-5
 - oop, 10-1
 - property, 10-5
- ARRAY VARIABLE
 - classes, 5-1
- asynchronous
 - I/O devices active object, 11-1
- BFILE class
 - DESTROY method, 8-3
 - FI_CLOSE method, 8-3
 - FI_OPEN method, 8-3
 - FI_READ method, 8-3
 - FL_REWIND method, 8-4
 - FL_SENSE_DATA method, 8-4
 - FL_SET_BUF_LEN method, 8-3
 - methods, 8-3
 - oop class, 8-2
- BINARY FILE
 - classes, 8-1
- bring
 - IPCS, 20-5
 - oop, 20-1
- BUZSND class
 - AO_CANCEL method, 13-6
 - AO_INIT method, 13-5
 - AO_QUEUE method, 13-6
 - AO_RUN method, 13-6
 - methods, 13-5
 - oop, 13-4
- cl_add
 - CLEANUP class function, 9-4
- CL_ADD
 - CLEANUP class method, 9-3
- cl_add_alloc
 - CLEANUP class function, 9-5
- cl_add_dyl
 - CLEANUP class function, 9-5
- cl_add_iochan
 - CLEANUP class function, 9-5
- cl_add_object
 - CLEANUP class function, 9-4
- cl_add_shared
 - CLEANUP class function, 9-5
- cl_clean_item
 - CLEANUP class function, 9-6
- CL_CLEAN_ITEM
 - CLEANUP class method, 9-3
- CL_CLEAN_LEVEL
 - CLEANUP class method, 9-4
- CL_INIT
 - CLEANUP class method, 9-3
- cl_remove
 - CLEANUP class function, 9-6
- CL_REMOVE
 - CLEANUP class method, 9-3
- CL_SET_LEVEL
 - CLEANUP class method, 9-4
- class
 - ACTIVE, 11-1
 - AIDLE, 12-1
 - ANIMATOR, 13-3
 - application manager definition, 10-4
 - application manager diagram, 10-4
 - application manager, 10-1
 - application manager property, 10-5
 - BFILE, 8-2
 - binary files, 8-1
 - BUZSND, 13-4
 - cleanup, 9-1
 - editable documents, 6-1
 - EPFLAT, 6-7
 - EPROOT, 6-2
 - EPSEG, 6-10
 - FACTIVE, 14-1
 - FCASY, 14-11
 - FCSYNC, 14-14
 - FMAN, 16-2
 - FMFMT, 16-13
 - FMMK, 16-12
 - FMSCAN, 16-14
 - FMSRC, 16-17
 - FMTARG, 16-18
 - FNODE, 14-8
 - FSCAN, 14-3
 - idle object, 12-1
 - IPCS, 19-2
 - LINKCL, 20-2
 - LINKSV, 20-4
 - PNODE, 15-3
 - PSEL, 15-4
 - PSELVAR, 15-2
 - resource files, 7-1
 - ROOT, 2-1
 - SCAN, 17-1
 - SERFILE, 8-11
 - SERVER, 19-4
 - SGBUF, 4-1
 - SYSTEM, 18-1
 - TIME, 3-1
 - TIMER, 13-1, 13-2
 - TLVDATA, 8-8
 - TLVFILE, 8-4
 - VAFIX, 5-9
 - VAFLAT, 5-14
 - variable array classes, 5-1
 - VAROOT, 5-3
 - VASEG, 5-16
 - VASTR, 5-11
 - VAXVAR, 5-18
 - VAXVARS, 5-21
- class diagrams
 - OLIB hierarchy, 1-3
 - OLIB library, 1-3
- classes
 - OLIB library overview, 1-1

-
- OLIB library using, 1-2
 - CLEANUP class
 - cl_add function, 9-4
 - CL_ADD method, 9-3
 - cl_add_alloc function, 9-5
 - cl_add_dyl function, 9-5
 - cl_add_iochan function, 9-5
 - cl_add_object function, 9-4
 - cl_add_shared function, 9-5
 - cl_clean_item function, 9-6
 - CL_CLEAN_ITEM method, 9-3
 - CL_CLEAN_LEVEL method, 9-4
 - CL_INIT method, 9-3
 - cl_remove function, 9-6
 - CL_REMOVE method, 9-3
 - CL_SET_LEVEL method, 9-4
 - DESTROY method, 9-3
 - functions convenience, 9-4
 - methods, 9-3
 - oop, 9-1
 - compute intensive tasks
 - AIDLE class, 12-1
 - DatCommandPtr
 - magic static, 10-11
 - DESTROY
 - ACTIVE class method, 11-3
 - BFILE class method, 8-3
 - CLEANUP class method, 9-3
 - EPFLAT class method, 6-8
 - IPCS class method, 19-3
 - LINKCL class method, 20-3
 - ROOT class method, 2-2
 - RSCFILE class method, 7-2
 - SERVER class method, 19-5
 - SGBUF class method, 4-2
 - VAROOT class method, 5-4
 - documents editable
 - classes, 6-1
 - DYL
 - OLIB library introduction, 1-1
 - editable documents class
 - oop, 6-1
 - EF_GRANULARITY
 - EPFLAT class method, 6-9
 - EF_SENSE_BUF
 - EPFLAT class method, 6-9
 - EP_ADD_PARA
 - EPROOT class method, 6-5
 - EP_BACK_CHARS
 - EPFLAT class method, 6-8
 - EPSEG class method, 6-11
 - EP_CAPACITY
 - EPFLAT class method, 6-9
 - EPROOT class method, 6-6
 - EP_CLEAR
 - EPFLAT class method, 6-9
 - EPSEG class method, 6-11
 - EP_COMPRESS
 - EPFLAT class method, 6-9
 - EPSEG class method, 6-11
 - EP_COPY_INDENT
 - EPROOT class method, 6-5
 - EP_COPY_TO_BACK
 - EPROOT class method, 6-5
 - EP_COPY_TO_FRONT
 - EPROOT class method, 6-5
 - EP_DELETE
 - EPFLAT class method, 6-9
 - EPSEG class method, 6-11
 - EP_EXTRACT
 - EPFLAT class method, 6-9
 - EPSEG class method, 6-11
 - EP_INIT
 - EPFLAT class method, 6-8
 - EPSEG class method, 6-10
 - EP_INSERT
 - EPFLAT class method, 6-8
 - EPSEG class method, 6-11
 - EP_MOD_CHARS
 - EPROOT class method, 6-6
 - EP_PARA_COUNT
 - EPROOT class method, 6-4
 - EP_PASTE
 - EPROOT class method, 6-6
 - EP_SCAN_BLOCK
 - EPROOT class method, 6-4
 - EP_SCAN_PARA
 - EPROOT class method, 6-4
 - EP_SCAN_WORD
 - EPROOT class method, 6-4
 - EP_SENSE_CHARS
 - EPFLAT class method, 6-8
 - EPSEG class method, 6-11
 - EP_SENSE_LEN
 - EPFLAT class method, 6-8
 - EPSEG class method, 6-11
 - EP_SENSE_TEXT
 - EPROOT class method, 6-6
 - EP_SET_TEXT
 - EPROOT class method, 6-3
 - EP_WORD COUNT
 - EPROOT class method, 6-4
 - EPFLAT class
 - DESTROY method, 6-8
 - EF_GRANULARITY method, 6-9
 - EF_SENSE_BUF method, 6-9
 - EP_BACK_CHARS method, 6-8
 - EP_CAPACITY method, 6-9
 - EP_CLEAR method, 6-9
 - EP_COMPRESS method, 6-9
 - EP_DELETE method, 6-9
 - EP_EXTRACT method, 6-9
 - EP_INIT method, 6-8
 - EP_INSERT method, 6-8
 - EP_SENSE_CHARS method, 6-8
 - EP_SENSE_LEN method, 6-8
 - methods, 6-8
 - oop class, 6-7
 - EPROOT class
 - EP_ADD_PARA method, 6-5
 - EP_CAPACITY method, 6-6
 - EP_COPY_INDENT method, 6-5
 - EP_COPY_TO_BACK method, 6-5
 - EP_COPY_TO_FRONT method, 6-5
 - EP_MOD_CHARS method, 6-6
 - EP_PARA_COUNT method, 6-4
 - EP_PASTE method, 6-6
 - EP_SCAN_BLOCK method, 6-4
-

- EP_SCAN_PARA method, 6-4
- EP_SCAN_WORD method, 6-4
- EP_SENSE_TEXT method, 6-6
- EP_SET_TEXT method, 6-3
- EP_WORD COUNT method, 6-4
- methods deferred, 6-6
- methods, 6-3
- oop class, 6-2
- EPSEG class
 - EP_BACK_CHARS method, 6-11
 - EP_CLEAR method, 6-11
 - EP_COMPRESS method, 6-11
 - EP_DELETE method, 6-11
 - EP_EXTRACT method, 6-11
 - EP_INIT method, 6-10
 - EP_INSERT method, 6-11
 - EP_SENSE_CHARS method, 6-11
 - EP_SENSE_LEN method, 6-11
 - methods, 6-10
 - oop class, 6-10
- error handling
 - OLIB library, 1-4
 - OLIB library panics, 1-5
- FA_CLOSE
 - FACTIVE class method, 14-3
 - FCASY class method, 14-13
 - FNODE class method, 14-10
 - FSCAN class method, 14-6
- FACTIVE class
 - AO_ABRUN method, 14-3
 - AO_CANCEL method, 14-2
 - AO_INIT method, 14-2
 - FA_CLOSE method, 14-3
 - methods, 14-2
 - oop, 14-1
- FC_OPEN
 - FCASY class method, 14-13
- FC_REQUEST_COMP
 - deferred method, 14-14
 - FCASY class method deferred, 14-14
 - FMSRC class method, 16-17
 - FMTARG class method, 16-18
- FC_WRITE
 - FCASY class method, 14-12
 - FCSYNC class method, 14-15
- FCASY class
 - AO_CANCEL method, 14-12
 - AO_QUEUE method, 14-12
 - AO_RUN method, 14-13
 - FA_CLOSE method, 14-13
 - FC_OPEN method, 14-13
 - FC_WRITE method, 14-12
 - methods deferred, 14-14
 - methods, 14-12
 - oop, 14-11
- FCASY sub-class
 - oop, 14-1
- FCSYNC class
 - AO_QUEUE method, 14-15
 - FC_WRITE method, 14-15
 - methods, 14-15
 - oop, 14-14
- FCSYNC sub-class
 - oop, 14-1
- FI_CLOSE
 - BFILE class method, 8-3
- FI_OPEN
 - BFILE class method, 8-3
 - TLVFILE class method, 8-6
- FI_READ
 - BFILE class method, 8-3
- file
 - active object, 14-1
- FILE_BINARY
 - classes, 8-1
- file I/O
 - active objects and, 11-1
- file lists
 - oop, 15-1
- file management
 - oop, 16-1
- file manager
 - example code, 16-19
- file scan local
 - oop, 17-1
- file server
 - active objects and, 11-1
- files
 - type-length-value, 8-4
- filing system
 - nodes, 14-8
- FL_COUNT
 - TLVFILE class method, 8-6
- FL_DELREC
 - TLVFILE class method, 8-7
- FL_READ_BY_TYPE
 - TLVFILE class method, 8-7
- FL_REPLACE
 - TLVFILE class method, 8-7
- FL_REWIND
 - BFILE class method, 8-4
 - TLVFILE class method, 8-6
- FL_SENSE_DATA
 - BFILE class method, 8-4
- FL_SENSE_REC
 - TLVFILE class method, 8-7
- FL_SET_BUF_LEN
 - BFILE class method, 8-3
- FL_SET_REC
 - TLVFILE class method, 8-6
- FL_WRITE_REC
 - TLVFILE class method, 8-6
- FMAN class
 - file management, 16-1
 - FMAN_ATTRIB method, 16-8
 - FMAN_CANCEL method, 16-4
 - FMAN_COMPLETE deferred method, 16-9
 - FMAN_COPY method, 16-5
 - FMAN_COPYDEV method, 16-7
 - FMAN_DELETE method, 16-5
 - FMAN_ERROR deferred method, 16-11
 - FMAN_FILEEXIST deferred method, 16-11
 - FMAN_FORMAT method, 16-8
 - FMAN_INFO method, 16-8
 - FMAN_INIT method, 16-4
 - FMAN_MAKE method, 16-7
 - FMAN_NAME method, 16-8

-
- FMAN_NEWNAME deferred method, 16-10
 - FMAN_REMOVE method, 16-7
 - FMAN_RENAME method, 16-6
 - FMAN_UPDATE deferred method, 16-11
 - methods deferred, 16-9
 - methods, 16-4
 - oop, 16-2
 - FMAN_ATTRIB
 - FMAN class method, 16-8
 - FMAN_CANCEL
 - FMAN class method, 16-4
 - FMAN_COMPLETE
 - FMAN class method deferred, 16-9
 - FMAN_COPY
 - FMAN class method, 16-5
 - FMAN_COPYDEV
 - FMAN class method, 16-7
 - FMAN_DELETE
 - FMAN class method, 16-5
 - FMAN_ERROR
 - FMAN class method deferred, 16-11
 - FMAN_FILEEXIST
 - FMAN class method deferred, 16-11
 - FMAN_FORMAT
 - FMAN class method, 16-8
 - FMAN_INFO
 - FMAN class method, 16-8
 - FMAN_INIT
 - FMAN class method, 16-4
 - FMAN_MAKE
 - FMAN class method, 16-7
 - FMAN_NAME
 - FMAN class method, 16-8
 - FMAN_NEWNAME
 - FMAN class method deferred, 16-10
 - FMAN_REMOVE
 - FMAN class method, 16-7
 - FMAN_RENAME
 - FMAN class method, 16-6
 - FMAN_UPDATE
 - FMAN class method deferred, 16-11
 - FMFMT class
 - AO_ABRUN method, 16-14
 - AO_RUN method, 16-13
 - methods, 16-13
 - oop, 16-13
 - FMFMT sub-class
 - oop, 16-1
 - FMMK class
 - AO_RUN method, 16-12
 - methods, 16-12
 - oop, 16-12
 - FMMK sub-class
 - oop, 16-1
 - FMSCAN class
 - AO_ABRUN method, 16-15
 - FS_DIRNAME method, 16-16
 - FS_END_DIRLIST method, 16-16
 - FS_FILENAME method, 16-15
 - FS_FSCAN_END method, 16-16
 - methods, 16-15
 - oop, 16-14
 - FMSCAN sub-class
 - oop, 16-1
 - FMSRC class
 - FC_REQUEST_COMP method, 16-17
 - methods, 16-17
 - oop, 16-17
 - FMSRC sub-class
 - oop, 16-1
 - FMTARG class
 - FC_REQUEST_COMP method, 16-18
 - methods, 16-18
 - oop, 16-18
 - FMTARG sub-class
 - oop, 16-1
 - FN_END_LIST
 - FNODE class method deferred, 14-10
 - PNODE class method, 15-4
 - FN_LIST
 - FNODE class method, 14-10
 - FN_NODENAME
 - FNODE class method deferred, 14-10
 - PNODE class method, 15-4
 - FNODE class
 - AO_QUEUE method, 14-9
 - AO_RUN method, 14-9
 - FA_CLOSE method, 14-10
 - FN_END_LIST deferred method, 14-10
 - FN_LIST method, 14-10
 - FN_NODENAME deferred method, 14-10
 - methods deferred, 14-10
 - methods, 14-9
 - oop, 14-8
 - FNODE sub-class
 - oop, 14-1
 - FS_DIRNAME
 - FMSCAN class method, 16-16
 - FSCAN class method deferred, 14-7
 - PSEL class method, 15-8
 - FS_END_DIRLIST
 - FMSCAN class method, 16-16
 - FSCAN class method deferred, 14-8
 - FS_FILENAME
 - FMSCAN class method, 16-15
 - FSCAN class method deferred, 14-7
 - PSEL class method, 15-8
 - FS_FSCAN
 - FSCAN class method, 14-6
 - FS_FSCAN_END
 - FMSCAN class method, 16-16
 - FSCAN class method deferred, 14-7
 - PSEL class method, 15-8
 - FS_MATCHNAME
 - FSCAN class method, 14-6
 - FSCAN class
 - AO_QUEUE method, 14-5
 - AO_RUN method, 14-5
 - FA_CLOSE method, 14-6
 - FS_DIRNAME deferred method, 14-7
 - FS_END_DIRLIST deferred method, 14-8
 - FS_FILENAME deferred method, 14-7
 - FS_FSCAN method, 14-6
 - FS_FSCAN_END method, 14-7
 - FS_MATCHNAME method, 14-6
 - methods deferred, 14-7
-

- methods, 14-5
- oop, 14-3
- FSCAN sub-class
 - oop, 14-1
- functions
 - CLEANUP class convenience, 9-4
- HWIM
 - active object, 11-1
 - application manager class, 10-1
 - idle object, 12-1
- I/O asynchronous
 - active object, 11-1
- idle object
 - class, 12-1
- IDLE OBJECT
 - class, 12-1
- idle time
 - AIDLE class computation, 12-3
- INTER-PROCESS COMMUNICATIONS
 - class, 19-1
- IP_ADD_SERVER
 - IPCS class method, 19-4
- IPCS
 - bring, 20-5
 - class, 19-1
 - link paste, 20-5
- IPCS class
 - AO_ABRUN method, 19-4
 - AO_CANCEL method, 19-3
 - AO_INIT method, 19-3
 - AO_QUEUE method, 19-3
 - AO_RUN method, 19-3
 - DESTROY method, 19-3
 - IP_ADD_SERVER method, 19-4
 - methods, 19-3
 - services, 19-2
- LC_GET_DATA
 - LINKCL class method, 20-3
- LC_START
 - LINKCL class method, 20-3
- LC_STOP
 - LINKCL class method, 20-3
- library
 - OLIB DYL introduction, 1-1
- link paste
 - IPCS, 20-5
- LINK PASTE
 - classes, 20-1
- LINKCL class
 - DESTROY class method, 20-3
 - LC_GET_DATA class method, 20-3
 - LC_START class method, 20-3
 - LC_STOP class method, 20-3
 - methods, 20-3
 - services, 20-2
- LINKSV class
 - LS_GET_DATA class method, 20-6
 - LS_SET_FORMAT class method, 20-6
 - methods deferred, 20-6
 - methods, 20-5
 - services, 20-4
 - SV_INIT class method, 20-5
 - SV_RUN class method, 20-5

- LOC
 - filing system node, 14-8, 17-1
- LOCS class
 - LS_FILENAME deferred method, 17-3
 - LS_MATCHNAME method, 17-3
 - LS_SCAN method, 17-3
 - methods deferred, 17-3
 - methods, 17-3
- LS_FILENAME
 - LOCS class method deferred, 17-3
- LS_GET_DATA
 - LINKSV class method deferred, 20-6
- LS_MATCHNAME
 - LOCS class method, 17-3
- LS_SCAN
 - LOCS class method, 17-3
- LS_SET_FORMAT
 - LINKSV class method deferred, 20-6
- magic static
 - DatCommandPtr, 10-11
- manager
 - application class, 10-1
- method function
 - OLIB long parameters, 1-3
 - OLIB prototypes, 1-2
- methods
 - ACTIVE class, 11-3
 - AIDLE class, 12-2
 - ANIMATOR class, 13-4
 - APPMAN class, 10-5
 - BFILE class, 8-3
 - BUZSND class, 13-5
 - CLEANUP class, 9-3
 - EPFLAT class, 6-8
 - EPROOT class deferred, 6-6
 - EPROOT class, 6-3
 - EPSEG class, 6-10
 - FACTIVE class, 14-2
 - FCASY class deferred, 14-14
 - FCASY class, 14-12
 - FCSYNC class, 14-15
 - FMAN class deferred, 16-9
 - FMAN class, 16-4
 - FMFMT class, 16-13
 - FMMK class, 16-12
 - FMSCAN class, 16-15
 - FMSRC class, 16-17
 - FMTARG class, 16-18
 - FNODE class deferred, 14-10
 - FNODE class, 14-9
 - FSCAN class deferred, 14-7
 - FSCAN class, 14-5
 - IPCS class, 19-3
 - LINKCL class, 20-3
 - LINKSV class deferred, 20-6
 - LINKSV class, 20-5
 - LOCS class deferred, 17-3
 - LOCS class, 17-3
 - PNODE class, 15-4
 - PSEL class deferred, 15-11
 - PSEL class, 15-7
 - PSELVAR class, 15-3
 - ROOT class, 2-2
 - RSCFILE class, 7-2

-
- SERFILE class, 8-13
 - SERVER class deferred, 19-6
 - SERVER class, 19-5
 - SGBUF class, 4-2
 - SYSTEM class, 18-2
 - TIME class, 3-3
 - TIMER class, 13-2
 - TLVDATA class deferred, 8-10
 - TLVDATA class, 8-9
 - TLVFILE class, 8-6
 - VAFIX class, 5-10
 - VAFLAT class, 5-15
 - VAROOT class deferred, 5-7
 - VAROOT class, 5-4
 - VASEG class, 5-17
 - VASTR class, 5-12
 - VAXVAR class, 5-20
 - VAXVARS class, 5-22
 - node
 - filing systems, 14-8
 - OLIB
 - class diagrams, 1-3
 - class hierarchy, 1-3
 - classes overview, 1-1
 - classes using, 1-2
 - error handling, 1-4
 - error numbers panics, 1-5
 - library introduction, 1-1
 - method function long parameters, 1-3
 - method function prototypes, 1-2
 - PLIB basis, 1-1
 - oop
 - ACTIVE class HWIM, 11-1
 - ACTIVE class methods, 11-3
 - active object priorities, 10-2
 - active object return value, 10-3
 - active object scheduling, 10-3
 - active object scheduling mechanism, 10-6
 - AIDLE class example, 12-3
 - AIDLE class HWIM, 12-1
 - AIDLE class idle time computation, 12-3
 - AIDLE class methods, 12-2
 - AIDLE class pause operation, 12-3
 - ANIMATOR class methods, 13-4
 - application manager class definition, 10-4
 - application manager class diagram, 10-4
 - application manager class HWIM, 10-1
 - application manager class property, 10-5
 - APPMAN class methods, 10-5
 - BFILE class, 8-2
 - BFILE class methods, 8-3
 - bring, 20-1
 - BUZSND class methods, 13-5
 - CLEANUP class functions convenience, 9-4
 - CLEANUP class methods, 9-3
 - EPFLAT class, 6-7
 - EPFLAT class methods, 6-8
 - EPROOT class, 6-2
 - EPROOT class deferred methods, 6-6
 - EPROOT class methods, 6-3
 - EPSEG class, 6-10
 - EPSEG class methods, 6-10
 - FACTIVE class, 14-1
 - FACTIVE class methods, 14-2
 - FCASY class, 14-11
 - FCASY class deferred methods, 14-14
 - FCASY class methods, 14-12
 - FCASY sub-class, 14-1
 - FCSYNC class, 14-14
 - FCSYNC class methods, 14-15
 - FCSYNC sub-class, 14-1
 - file active object, 14-1
 - file lists, 15-1
 - file management, 16-1
 - file manager example code, 16-19
 - file scan local, 17-1
 - FMAN class, 16-2
 - FMAN class deferred methods, 16-9
 - FMAN class methods, 16-4
 - FMFMT class, 16-13
 - FMFMT class methods, 16-13
 - FMFMT sub-class, 16-1
 - FMMK class, 16-12
 - FMMK class methods, 16-12
 - FMMK sub-class, 16-1
 - FMSCAN class, 16-14
 - FMSCAN class methods, 16-15
 - FMSCAN sub-class, 16-1
 - FMSRC class, 16-17
 - FMSRC class methods, 16-17
 - FMSRC sub-class, 16-1
 - FMTARG class, 16-18
 - FMTARG class methods, 16-18
 - FMTARG sub-class, 16-1
 - FNODE class, 14-8
 - FNODE class deferred methods, 14-10
 - FNODE class methods, 14-9
 - FNODE sub-class, 14-1
 - FSCAN class, 14-3
 - FSCAN class deferred methods, 14-7
 - FSCAN class methods, 14-5
 - FSCAN sub-class, 14-1
 - inter-process communications, 19-1
 - IPCS, 19-1
 - IPCS bring, 20-5
 - IPCS class, 19-2
 - IPCS class methods, 19-3
 - IPCS link paste, 20-5
 - link paste, 20-1
 - LINKCL class, 20-2
 - LINKCL class methods, 20-3
 - LINKSV class, 20-4
 - LINKSV class deferred methods, 20-6
 - LINKSV class methods, 20-5
 - LOCS class deferred methods, 17-3
 - LOCS class methods, 17-3
 - PNODE class, 15-3
 - PNODE class methods, 15-4
 - PSEL class, 15-4
 - PSEL class deferred methods, 15-11
 - PSEL class methods, 15-7
 - PSELVAR class, 15-2
 - PSELVAR class methods, 15-3
 - ROOT class, 2-1
 - ROOT class DESTROY method, 2-2
 - ROOT class methods, 2-2
 - RSCFILE class methods, 7-2
 - SCAN class, 17-1
-

- SERFILE class, 8-11
- SERFILE class methods, 8-13
- SERVER class, 19-4
- SERVER class deferred methods, 19-6
- SERVER class methods, 19-5
- SGBUF class, 4-1
- SGBUF class methods, 4-2
- SYSTEM class, 18-1
- SYSTEM class methods, 18-2
- system services, 18-1
- TIME class, 3-1
- TIME class methods, 3-3
- TIMER active object, 13-1
- TIMER class, 13-1
- TIMER class methods, 13-2
- TLVDATA class, 8-8
- TLVDATA class deferred methods, 8-10
- TLVDATA class methods, 8-9
- TLVFILE class, 8-4
- TLVFILE class methods, 8-6
- type-length-value files, 8-4
- VAFIX class, 5-9
- VAFIX class methods, 5-10
- VAFLAT class, 5-14
- VAFLAT class methods, 5-15
- VAROOT class, 5-3
- VAROOT class deferred methods, 5-7
- VAROOT class methods, 5-4
- VASEG class, 5-16
- VASEG class methods, 5-17
- VASTR class, 5-11
- VASTR class methods, 5-12
- VAXVAR class, 5-18
- VAXVAR class methods, 5-20
- VAXVARS class, 5-21
- VAXVARS class methods, 5-22
- panics
 - OLIB error numbers, 1-5
- pause operation
 - AIDLE example, 12-3
- PLIB
 - OLIB relationship to, 1-1
- PNODE class
 - AO_ABRUN method, 15-4
 - FN_END_LIST method, 15-4
 - FN_NODENAME method, 15-4
 - methods, 15-4
 - oop, 15-3
- PS_ASCEND_PATH
 - PSEL class method, 15-9
- PS_DESCEND_PATH
 - PSEL class method, 15-9
- PS_DRIVES
 - PSEL class method, 15-10
- PS_GET_FILE
 - PSEL class method, 15-8
- PS_GETTAG
 - PSEL class method, 15-10
- PS_NEW_LIST
 - PSEL class method deferred, 15-11
- PS_ORDER
 - PSEL class method, 15-10
- PS_SELECT_DIRENTRY
 - PSEL class method, 15-9
- PS_SENSE_FILENAME
 - PSEL class method, 15-9
- PS_SET_PATH
 - PSEL class method, 15-9
- PS_SETTAG
 - PSEL class method, 15-10
- PSEL class
 - AO_ABRUN method, 15-7
 - AO_CANCEL method, 15-7
 - AO_INIT method, 15-7
 - FS_DIRNAME method, 15-8
 - FS_FILENAME method, 15-8
 - FS_FSCAN_END method, 15-8
 - methods deferred, 15-11
 - methods, 15-7
 - oop, 15-4
 - PS_ASCEND_PATH method, 15-9
 - PS_DESCEND_PATH method, 15-9
 - PS_DRIVES method, 15-10
 - PS_GET_FILE method, 15-8
 - PS_GETTAG method, 15-10
 - PS_NEW_LIST deferred method, 15-11
 - PS_ORDER method, 15-10
 - PS_SELECT_DIRENTRY method, 15-9
 - PS_SENSE_FILENAME method, 15-9
 - PS_SET_PATH method, 15-9
 - PS_SETTAG method, 15-10
- pselvar
 - file lists, 15-1
- PSELVAR class
 - methods, 15-3
 - oop, 15-2
 - VA_TEST method, 15-3
- REM
 - filing system node, 14-8
- RESOURCE FILES
 - class, 7-1
- ROM
 - filing system node, 14-8, 17-1
- ROOT class
 - DESTROY method, 2-2
 - methods, 2-2
 - oop superclass, 2-1
- RS_INIT
 - RSCFILE class method, 7-2
- RS_READ
 - RSCFILE class method, 7-2
- RS_READ_BUF
 - RSCFILE class method, 7-2
- RSCFILE class
 - DESTROY method, 7-2
 - methods, 7-2
 - RS_INIT method, 7-2
 - RS_READ method, 7-2
 - RS_READ_BUF method, 7-2
- SB_ALLOCSEG
 - SGBUF class method, 4-4
- SB_BACKPOINT
 - SGBUF class method, 4-4
- SB_COMPRESS
 - SGBUF class method, 4-4
- SB_COUNT
 - SGBUF class method, 4-4

-
- SB_DELETE
 - SGBUF class method, 4-3
 - SB_EXTRACT
 - SGBUF class method, 4-4
 - SB_INIT
 - SGBUF class method, 4-2
 - SB_INSERT
 - SGBUF class method, 4-3
 - SB_POINT
 - SGBUF class method, 4-3
 - SCAN class
 - file scan local, 17-1
 - oop, 17-1
 - SERFILE class
 - methods, 8-13
 - oop class, 8-11
 - TD_RESET method, 8-13
 - TD_SENSE_ITEM method, 8-14
 - TD_SET_FILE method, 8-13
 - TD_SET_ITEM method, 8-13
 - serial port
 - parameter .trm file, 8-11
 - SERVER class
 - DESTROY method, 19-5
 - methods deferredoop, 19-6
 - methods, 19-5
 - services, 19-4
 - SV_ABRUN method, 19-5
 - SV_INIT method, 19-5
 - SV_RUN deferred method, 19-6
 - SGBUF class
 - DESTROY method, 4-2
 - methods, 4-2
 - oop class, 4-1
 - SB_ALLOCSEG method, 4-4
 - SB_BACKPOINT method, 4-4
 - SB_COMPRESS method, 4-4
 - SB_COUNT method, 4-4
 - SB_DELETE method, 4-3
 - SB_EXTRACT method, 4-4
 - SB_INIT method, 4-2
 - SB_INSERT method, 4-3
 - SB_POINT method, 4-3
 - sub-class
 - FCASY, 14-1
 - FCSYNC, 14-1
 - FMFMT, 16-1
 - FMMK, 16-1
 - FMSCAN, 16-1
 - FMSRC, 16-1
 - FMTARG, 16-1
 - FNODE, 14-1
 - FSCAN, 14-1
 - SV_ABRUN
 - SERVER class method, 19-5
 - SV_INIT
 - LINKSV class method, 20-5
 - SERVER class method, 19-5
 - SV_RUN
 - LINKSV class method, 20-5
 - SERVER class method deferred, 19-6
 - SY_EXEC_OPEN
 - SYSTEM class method, 18-3
 - SY_ICON_POS
 - SYSTEM class method, 18-2
 - SY_INIT
 - SYSTEM class method, 18-2
 - SY_LINK_PASTE
 - SYSTEM class method, 18-3
 - SY_LINK_SERVER
 - SYSTEM class method, 18-3
 - SYSTEM class
 - methods, 18-2
 - services, 18-1
 - SY_EXEC_OPEN method, 18-3
 - SY_ICON_POS method, 18-2
 - SY_INIT method, 18-2
 - SY_LINK_PASTE method, 18-3
 - SY_LINK_SERVER method, 18-3
 - SYSTEM SERVICES
 - class, 18-1
 - TD_CHANGED
 - TLVDATA class method, 8-9
 - TD_LOAD_ITEM
 - TLVDATA class method, 8-10
 - TD_OPEN
 - TLVDATA class method, 8-9
 - TD_RESET
 - SERFILE class method, 8-13
 - TLVDATA class method, 8-10
 - TD_SAVE
 - TLVDATA class method, 8-9
 - TD_SAVE_ITEM
 - TLVDATA class method, 8-10
 - TD_SENSE_ITEM
 - SERFILE class method, 8-14
 - TLVDATA class method deferred, 8-10
 - TD_SET_FILE
 - SERFILE class method, 8-13
 - TLVDATA class method deferred, 8-10
 - TD_SET_ITEM
 - SERFILE class method, 8-13
 - TLVDATA class method deferred, 8-10
 - TIME class
 - methods, 3-3
 - oop class, 3-1
 - TO_ADD_DAYS method, 3-5
 - TO_ADD_MONTHS method, 3-5
 - TO_ADD_SECS method, 3-4
 - TO_ADD_YEARS method, 3-5
 - TO_GET_SYSDAT method, 3-7
 - TO_SENSE method, 3-4
 - TO_SENSE_FORMAT method, 3-6
 - TO_SET method, 3-3
 - TO_SET_FORMAT method, 3-6
 - TIMER class
 - active object, 13-1
 - AO_INIT method, 13-2
 - AO_QUEUE method, 13-2
 - methods, 13-2
 - oop, 13-1, 13-2
 - TM_QABSOLUTE method, 13-3
 - TLVDATA class
 - methods deferred, 8-10
 - methods, 8-9
 - oop class, 8-8
 - TD_CHANGED method, 8-9
-

- TD_LOAD_ITEM method, 8-10
- TD_OPEN method, 8-9
- TD_RESET method, 8-10
- TD_SAVE method, 8-9
- TD_SAVE_ITEM method, 8-10
- TD_SENSE_ITEM method deferred, 8-10
- TD_SET_FILE method deferred, 8-10
- TD_SET_ITEM method deferred, 8-10
- TLVFILE class
 - FI_OPEN method, 8-6
 - FL_COUNT method, 8-6
 - FL_DELREC method, 8-7
 - FL_READ_BY_TYPE method, 8-7
 - FL_REPLACE method, 8-7
 - FL_REWIND method, 8-6
 - FL_SENSE_REC method, 8-7
 - FL_SET_REC method, 8-6
 - FL_WRITE_REC method, 8-6
 - methods, 8-6
 - oop class, 8-4
- TM_QABSOLUTE
 - TIMER class method, 13-3
- TO_ADD_DAYS
 - TIME class method, 3-5
- TO_ADD_MONTHS
 - TIME class method, 3-5
- TO_ADD_SECS
 - TIME class method, 3-4
- TO_ADD_YEARS
 - TIME class method, 3-5
- TO_GET_SYSDAT
 - TIME class method, 3-7
- TO_SENSE
 - TIME class method, 3-4
- TO_SENSE_FORMAT
 - TIME class method, 3-6
- TO_SET
 - TIME class method, 3-3
- TO_SET_FORMAT
 - TIME class method, 3-6
- type-length-value
 - files, 8-4
- VA_APPEND
 - VAROOT class method, 5-4
- VA_CAPACITY
 - VAFLAT class method, 5-15
 - VAROOT class method deferred, 5-8
 - VASEG class method, 5-17
 - VASTR class method, 5-13
- VA_COMPARE
 - VAROOT class method, 5-5
- VA_COMPRESS
 - VAFLAT class method, 5-15
 - VAROOT class method deferred, 5-8
 - VASEG class method, 5-17
 - VASTR class method, 5-13
- VA_COPY
 - VAFIX class method, 5-10
 - VAROOT class method deferred, 5-7
 - VASTR class method, 5-14
 - VAXVAR class method, 5-21
- VA_COUNT
 - VAROOT class method, 5-4
- VA_DELETE
 - VAROOT class method, 5-5
- VA_DELETITEM
 - VAFLAT class method, 5-15
 - VAROOT class method deferred, 5-8
 - VASEG class method, 5-17
 - VASTR class method, 5-13
 - VAXVAR class method, 5-20
- VA_FINDISQ
 - VAROOT class method, 5-6
- VA_INIT
 - VAFLAT class method, 5-15
 - VAROOT class method deferred, 5-8
 - VASEG class method, 5-17
 - VASTR class method, 5-12
 - VAXVAR class method, 5-20
- VA_INSERT
 - VAROOT class method, 5-4
- VA_INSERTISQ
 - VAROOT class method, 5-6
- VA_INSERTTM
 - VAFLAT class method, 5-15
 - VAROOT class method deferred, 5-8
 - VASEG class method, 5-18
 - VASTR class method, 5-13
 - VAXVAR class method, 5-20
- VA_KEY
 - VAROOT class method, 5-5
- VA_PBUF
 - VAFLAT class method, 5-16
 - VAROOT class method deferred, 5-9
 - VASEG class method, 5-18
 - VASTR class method, 5-14
 - VAXVAR class method, 5-21
- VA_PREC
 - VAFLAT class method, 5-16
 - VAROOT class method deferred, 5-9
 - VASEG class method, 5-18
 - VASTR class method, 5-13
- VA_RECLEN
 - VAFIX class method, 5-10
 - VAROOT class method deferred, 5-7
 - VASTR class method, 5-13
 - VAXVAR class method, 5-21
- VA_REPLACE
 - VAFIX class method, 5-10
 - VAROOT class method, 5-7
 - VAXVAR class method, 5-21
- VA_RESET
 - VAROOT class method, 5-7
- VA_SEARCH
 - VAROOT class method, 5-6
- VA_SORT
 - VAROOT class method, 5-6
- VA_SWAP
 - VAFIX class method, 5-10
 - VAROOT class method deferred, 5-7
- VA_TEST
 - PSELVAR class method, 15-3
 - VAROOT class method, 5-5
 - VAXVAR class method, 5-20
- VAFIX class
 - methods, 5-10
 - oop class, 5-9

- VA_COPY method, 5-10
- VA_RECLLEN method, 5-10
- VA_REPLACE method, 5-10
- VA_SWAP method, 5-10
- VAFLAT class
 - methods, 5-15
 - oop class, 5-14
 - VA_CAPACITY method, 5-15
 - VA_COMPRESS method, 5-15
 - VA_DELETETM method, 5-15
 - VA_INIT method, 5-15
 - VA_INSERTTM method, 5-15
 - VA_PBUF method, 5-16
 - VA_PREC method, 5-16
- VARIABLE ARRAY
 - classes, 5-1
- VAROOT class
 - DESTROY method, 5-4
 - methods deferred, 5-7
 - methods, 5-4
 - oop class, 5-3
 - VA_APPEND method, 5-4
 - VA_CAPACITY method deferred, 5-8
 - VA_COMPARE method, 5-5
 - VA_COMPRESS method deferred, 5-8
 - VA_COPY method deferred, 5-7
 - VA_COUNT method, 5-4
 - VA_DELETE method, 5-5
 - VA_DELETETM method deferred, 5-8
 - VA_FINDISQ method, 5-6
 - VA_INIT method deferred, 5-8
 - VA_INSERT method, 5-4
 - VA_INSERTISQ method, 5-6
 - VA_INSERTTM method deferred, 5-8
 - VA_KEY method, 5-5
 - VA_PBUF method deferred, 5-9
 - VA_PREC method deferred, 5-9
 - VA_RECLLEN method deferred, 5-7
 - VA_REPLACE method, 5-7
 - VA_RESET method, 5-7
 - VA_SEARCH method, 5-6
 - VA_SORT method, 5-6
 - VA_SWAP method deferred, 5-7
 - VA_TEST method, 5-5
- VASEG class
 - methods, 5-17
 - oop class, 5-16
 - VA_CAPACITY method, 5-17
 - VA_COMPRESS method, 5-17
 - VA_DELETETM method, 5-17
 - VA_INIT method, 5-17
 - VA_INSERTTM method, 5-18
 - VA_PBUF method, 5-18
 - VA_PREC method, 5-18
- VASTR class
 - methods, 5-12
 - oop class, 5-11
 - VA_CAPACITY method, 5-13
 - VA_COMPRESS method, 5-13
 - VA_COPY method, 5-14
 - VA_DELETETM method, 5-13
 - VA_INIT method, 5-12
 - VA_INSERTTM method, 5-13
 - VA_PBUF method, 5-14
 - VA_PREC method, 5-13
 - VA_RECLLEN method, 5-13
- VAXVAR class
 - methods, 5-20
 - oop class, 5-18
 - VA_COPY method, 5-21
 - VA_DELETETM method, 5-20
 - VA_INIT method, 5-20
 - VA_INSERTTM method, 5-20
 - VA_PBUF method, 5-21
 - VA_RECLLEN method, 5-21
 - VA_REPLACE method, 5-21
 - VA_TEST method, 5-20
- VAXVARS class
 - methods, 5-22
 - oop class, 5-21