SUN/233.4

Starlink Project
Starlink User Note 233.4

Tim Jenness, Frossie Economou, Brad Cavanagh
Joint Astronomy Centre, Hilo, Hawaii

June 2004

ORAC-DR – Programmer's Guide 4.1-0

Abstract

ORAC-DR is a general purpose automatic data reduction pipeline environment. This document describes how to modify data reduction recipes and how to add new instruments. For a general overview of ORAC-DR see SUN/230. For specific information on how to reduce the data for a particular instrument, please consult the appropriate ORAC-DR instrument guide.

Contents

1 Introduction

ORAC-DR is a flexible and modular pipeline developed by the Joint Astronomy Centre for the on-line reduction of data from infrared instruments. It is part of the UKIRT ORAC project. Additionally it is used for the reduction of data from SCUBA on the JCMT.

2 Overview

One of the main design goals of the ORAC system was for it to be modular. Figure 1 shows the basic components of the system. In theory, each component can be replaced without affecting the other systems.[1]

This document provides information on writing recipes and for adding support for new instruments to the pipeline.

3 Recipes

Recipes in ORAC-DR consist of a series of data reduction steps (primitives) containing instructions for the reduction of data. In general, recipes are different for each instrument supported and are keyed to the specific observing mode used to take the data.

An example recipe may look something like:

```
=head1 NAME

RECIPE_NAME - Short description of what the recipe does

=head1 DESCRIPTION

Long description of what recipe does.

=head1 NOTES

Some notes associated with the recipe.

=head1 SEE ALSO

List of related recipes.

=head1 AUTHORS

List of authors
```

[1]Although in practice, changing the algorithm engine usually involves a change in the primitive and possibly a change in the messaging layer!

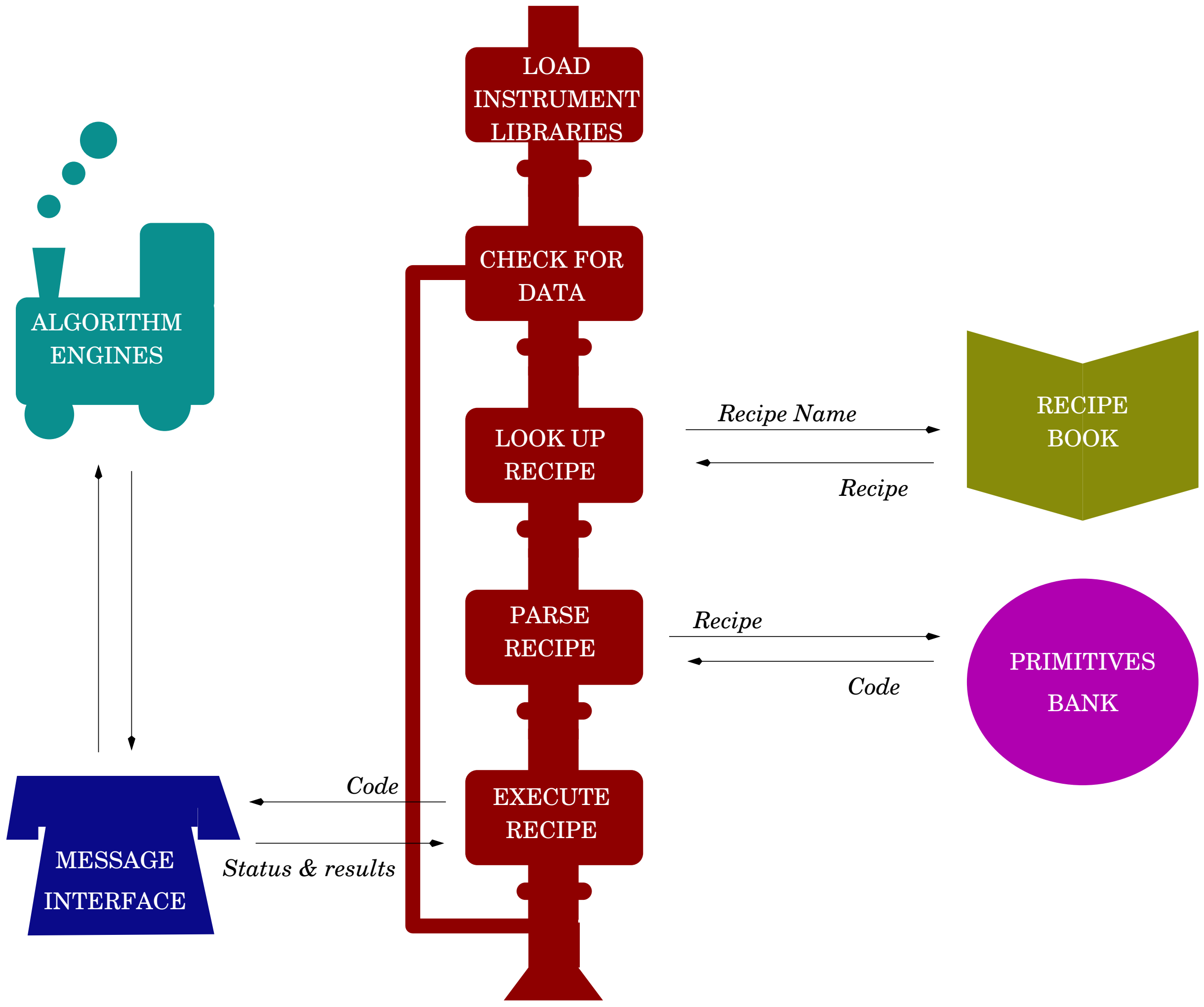

Figure 1: Outline of the modularity of ORAC-DR

```
=cut

_INSTRUMENT_HELLO_
# A comment
_STEP_ONE_
_STEP_TWO_  ARG1=number ARG2=string
_STEP_THREE_
_TIDY_RECIPE_
```

This recipe illustrates all the important features of a recipe:

- The recipe must contain documentation in the Perl POD syntax (see e.g. the PERLPOD manpage). This documentation is used by the `oracman` command and for the automated documentation system. At the very least, recipe documentation should include **NAME** and **DESCRIPTION** fields. The **NAME** field should use the standard Perl format of

  ```
  RECIPE_NAME - purpose
  ```

  so that the POD translators can correctly determine this information when generating LaTeX and HTML code.

- To simplify the readability of recipes for non-programmers and to separate the recipe language from the programming language used to implement the primitives, *no computer code should be visible in the recipe*.[2]

- Recipes are in plain text, and are easily modified by both support scientists and users with the aid of a text editor. This also means that is should be easy to support a GUI based drag-and-drop-type recipe builder at a future date.

- For most instruments, it is required that a **HELLO** primitive be included at the start of every recipe. This is used to guarantee that certain initialization steps are always executed. See individual instrumentation documentation to see whether a **HELLO** primitive is required.

- The comment character is a #. All text after a # is ignored by the recipe parser.

- Configuration options can be passed into primitives by the use of a `KEYWORD=value` syntax. The recipe parser automatically converts these values into arguments suitable for the primitives.

- A **TIDY** primitive is required at the end of most recipes. This can be used to remove intermediate files at the end of a recipe and any other tidying operation It usually calls the `_DELETE_TEMP_FILES_` primitive. It must make sure that files required for subsequent group processing are not deleted.

Here is a more concrete example, the UFTI `QUADRANT_JITTER` recipe (without the pod sections and only minimal comments):

[2] ORAC-DR does not *enforce* this rule though and it is possible to include Perl constructs in private recipes and for testing. It should not be done for production code.

```
# Initialisation
_IMAGING_HELLO_
_CREATE_WCS_
_QUADRANT_JITTER_HELLO_
_QUADRANT_JITTER_STEER_
# Calibration
_MASK_BAD_PIXELS_
_SUBTRACT_DARK_
_FLAT_FIELD_QUADRANT_JITTER_ MASK=1
# Mosaicking
_GENERATE_OFFSETS_QUADRANT_JITTER_ PERCENTILE=99 COMPLETE=0.4 MINPIX=12
_MAKE_MOSAIC_QUADRANT_OPTIMISED_ RESAMPLE=1 INT_METHOD=linint FILLBAD=1
# Cleanup
_QUADRANT_JITTER_TIDY_
```

3.1 Recipe Names

The names of recipes should be descriptive. Short obtuse names are not recommended since it should be obvious to the observer which recipe is associated with which observing mode. In general, recipe names are read from the file header, the location of which is specified by the instrument specific classes in the ORAC-DR. If required though, the recipe can be specified on the command line although this can cause problems if the user specifies the wrong observation numbers.

3.2 Recipe locations

ORAC-DR searches in two locations for the recipe files. First, the directory specified by the `ORAC_RECIPE_DIR` environment variable, if defined, is searched. This allows users of the pipeline to modify a standard recipe without editing the original. If the variable is not set or the recipe can not be found, the default location of `$ORAC_DIR/recipes/$ORAC_INSTRUMENT` is searched (although ORAC-DR can sometimes be used to modify the value of `ORAC_INSTRUMENT` during initialization). The pipeline aborts if the recipe cannot be located.

4 Primitives

ORAC-DR primitives contain information for the manipulation of data in a given state. They are written using object-oriented techniques, manipulating objects associated with the individual data frames as well as groups of observations. The steps that involve actual processing of data (as opposed to housekeeping etc. tasks) are done via a messaging request to an algorithm engine resident in memory.

Because primitives always manipulate objects associated with the pipeline, they are order-ignorant, i.e. no assumptions are made about the file number, file name, or filename convention. These behaviours are all handled by instrument-specific classes, thus allowing the possibility of changing these conventions for an instrument without changing code and of re-use of primitives elsewhere in different recipes for different instruments.

Additionally, it is possible for primitives to contain instructions to include other primitives. An arbritrary limit of 10 levels is imposed by the software to provide protection against recursion. This can easily be extended if required but currently no recipes include primitives at depths greater than 5. The recipe parser recognises primitive inclusion directives by matching the pattern `^\s*_` in non-pod sections, i.e. an underscore as the first blank character on a line. The primitive directive should look exactly as it would if found in a recipe, i.e.:

```
_SOME_PRIMITIVE_ ARG=arg  ARG2=$a  ARG3=$b
```

and not

```
_SOME_PRIMITIVE_(ARG=arg, ARG2=$a, ARG3=$b);
```

as would normally be expected for Perl. One difference from the recipe level[3] is that Perl scalar variables can be used to pass values into the primitive rather than having to hard-code a specific value. The current recipe parser does not yet allow complex data structures (arrays, hashes, objects, references) to be passed into primitives. This is because of a desire to enforce simple interfaces to all primitives regardless of location rather than an inability of the parser to be modified to support it. If there is demand for it, this feature could easily be added.

The following variables are available to all primitives:

$Frm

Object of class `ORAC::Frame` (or a subclass thereof). This object contains information about the *current* frame being processed by the pipeline. Methods are provided for accessing the current filename and the header information.

$Grp

Object of class `ORAC::Group` (or a subclass thereof). This object contains information about the *current* group being processed by the pipeline. In ORAC-DR a group is thought of as an array of frame objects. Methods are provided for accessing the current filename and current group members. The current frame will be a member of the current group. The usual behaviour is that the group object will contain the frames processed so far that are part of the group, in which case the last member of the group is the current frame. Alternatively, it is possible for the group object to know about all members of the group regardless of which have been processed already, in which case the current frame may not necessarily be the last frame of the group. The latter behaviour is controlled by the `-batch` switch in ORAC-DR and can be used by primitive writers to delay group processing until the current frame is the last member of the group (a group method is provided to determine this). This is only relevant for processing off-line but can significantly reduce run time of certain recipes. Currently, SCUBA is the only instrument that supports the batch option.

$Cal

This provides access to the instrument's calibration system. This object can return information such as the dark and flats to use for infrared data or the current sky opacity for SCUBA. Extensive use is made of index files and some primitives and recipes must be

[3]purely because the recipe level does not contain code

responsible for filing calibration data with this object so that later recipes can access the correct information. The behaviour of this object is completely instrument specific, in general very few methods are inherited by subclasses.

$Display

An object of class `ORAC::Display`. Used by primitive writers to send data display commands to the display subsystem. Note that sending a display command does not necessarily result in anything being displayed. The display system itself is not discussed in this document [see the internals document].

%Mon

A hash containing `ORAC::Msg` objects. These objects are used to send messages to the algorithm engines. The hash keys will describe which algorithm engine is to be contacted. The instrument interface should describe which tasks are available to programmers. Methods are provided to send messages to algorithm engines (always waiting for the reply before continuing) and for retrieving parameter values.

$ORAC_PRIMITIVE

This is the name of the current primitive. Usually used for debug messages. Since this variable has a scope of the current primitive only, if other primitives are included from within a primitive a new variable will be defined in the scope of the included primitive. This will generate warnings if the `-w` switch is turned on since the current variable will mask the variable defined in the scope of the parent primitive (as desired).

%_PRIMITIVE_NAME_

Hash containing the arguments available to the current primitive. The name of the hash is the same as the name of the primitive (i.e. the hash is not named `_PRIMITIVE_NAME_` explicitly).

All primitives (and in fact all of ORAC-DR) are written with the `strict` pragma turned on (see `perldoc strict` for more information) primarily to force a declared scope for all variables. All primitives are evaluated by the Perl interpreter as

```
preamble added by the parser
{
  preamble added by the parser in limited scope
  _PRIMITIVE_
}
```

such that each primitive is in a different block to all other primitives[4]. The only global variables used by primitives should be those supplied by the pipeline, all other variables should be lexical (i.e. using the Perl `my` declaration) and will therefore be freed at the end of the primitive. Passing complex information between primitives must be achieved by other means and is discussed in §5.3.

All recipes are evaluated in the `ORAC::Basic` namespace. This fact should not be used by any primitives and it is not guaranteed that this namespace will be used in future releases of ORAC-DR. It can always be assumed that functions and methods from the following modules will be visible to all primitives:

[4]unless the primitive is nested inside another primitive

ORAC::Print

Provides the `orac_print`, `orac_err` and `orac_warn` commands. These routines route messages to the correct output systems (an X-window, the screen, a log file etc) specified by the user. The standard perl functions `print` and `warn` should not be used for user messages except during development since the user has no control of where to send them.

ORAC::LogFile

Used to write log files (see §5.1).

ORAC::Constants

Provides access to the standard ORAC-DR constants. The most important are `ORAC__OK` and `ORAC__ERROR` (note that they are Perl constants, not variables).

ORAC::TempFile

Use to generate temporary files (see §5.2).

ORAC::General

Functions that are useful but are not necessarily a standard part of Perl. Examples are `max()`, `min()` and `log10()`.

5 Writing a Primitive

In order to write a valid primitive certain steps must be adhered to in order to ensure that subsequent primitives have the correct information.

It assumes knowledge of the following Perl concepts: using Perl objects, lexical variables, Perl data structures.

Here is an example primitive showing the basic principles:

```
=head1 NAME

_PRIMITIVE_NAME_ - short description

=head1 DESCRIPTION

Long description

=head1 ARGUMENTS

=over 4

=item ARG1

Description of possible values of ARG1

etc...

=back
```

```
20
21  =head1 TASKS
22
23  List of external tasks required by the primitive
24
25  =head1 OUTPUT FILES
26
27  Output suffix for the display system.
28
29  etc, AUTHORS, COPYRIGHT.....
30
31  =cut
32
33  # Read arguments or use default values
34  my $arg1 = ( exists $_PRIMITIVE_NAME{ARG1} ?
35               $_PRIMITIVE_NAME{ARG1} : 5);
36
37  # Loop over all sub frames
38  foreach my $i (1..$Frm->nfiles) {
39
40    # Get the input and output filename
41    my ($in, $out) = $Frm->inout('_sfx', $i);
42
43    # Read some value from the frame FITS header
44    my $value = $Frm->hdr('KEYWORD');
45
46    # Combine the in, out and value into options for the task
47    # DEPENDS ON ALGORITHM ENGINE
48    my $options = "IN=$in OUT=$out SWITCH=$value";
49
50    # Run the algorithm engine
51    $Mon{'task'}->obeyw('TASK',$options);
52
53    # Retrieve an answer from a parameter
54    ($ORAC_STATUS, $result) = $Mon{'task'}->get('TASK','PARAMETER');
55
56    # Print the result
57    orac_print "Result from primitive $ORAC_PRIMITIVE = $result\n";
58
59    # Update the frame object so that the next primitive
60    # gets the correct input file name
61    $Frm->file($i, $out);
62
63  }
64
65  # Ask the display system to display the frame
66  $Display->display_data($Frm) if defined $Display
67
```

The following should be noted:

- Primitives are written in Perl and all variables are lexicals (using `my`).
- As for recipes, the first few lines (1–31) are the documentation for the primitive in pod format. In addition to the fields used for recipe headers, three new fields are required for

primitives, **ARGUMENTS** to describe the configuration options, **TASKS** to list external dependencies and **OUTPUT FILES** to list the suffix of any output data files (for use with the display system) and any log files that may be written.

- Supplied arguments can be read from the _PRIMITIVE_NAME_ hash. Lines 33–35 check for the existence of the `ARG1` key in the hash and read it if it exists else a default value is copied in. Note the use of `exists` rather than `defined` for checking hash contents. If `defined` is used the key would automatically be created in the hash and set to a value of `undef` whereas `exists` simply looks for the key in the hash without creating it. In general, this is the more correct behaviour as it can distinguish between the key not being there at all (i.e. never set) and the key being set explicitly to `undef`[5].

  In some current primitives the following may be found for reading arguments:

  ```
  my $arg1 = ( $_PRIMITIVE_NAME_{ARG1} || $default);
  ```

  This works in most normal cases but will fail if the value of the argument is desired to be '0' since that evaluates to false and will cause the default to be returned rather than a '0'. This construct also causes the key to be created even if no argument was ever supplied (known as *auto-vivification*).

- Line 38 starts a loop over all the sub-frames present in the frame. The need for such a loop depends on the individual instrument. UFTI, for example, only ever creates a single data frame per disk file and so will not require the loop. SCUBA can take data for multiple wavelengths, and MICHELLE can store multiple integrations per data file, so this loop would ensure that each wavelength/integration is processed in turn.

- Line 41 retrieves the name of the current input file and supplies a valid output filename based on the supplied suffix, "`_sfx`". Note that this command accepts a number as an optional argument. This can be used to connect the file name with the sub-frame (a Frame object can store multiple current filenames).

- Line 44 simply retrieves a value from the header. The `hdr()` method can also be used to set header values (but does not change the header on disk).

- Lines 46–51 send a message to an algorithm engine using a messaging object stored in the `%Mon` hash. The options string depends on the task at the other end of the message bus and will therefore need to be changed if the algorithm engine is changed. An important point is that error checking code is automatically inserted into the recipe when `->obeyw` is found in a line that is not preceeded by an equals sign or a comment. This means that line 51 is translated to:

  ```
  {
    my $OBEYW_STATUS = $Mon{'task'}->obeyw('TASK', $options);
    if ($OBEYW_STATUS != ORAC__OK) {
       < ERROR MESSAGE CONSTRUCTED AND PRINTED WITH orac_err>
       return $OBEYW_STATUS;
    }
  }
  ```

[5]Although it is not possible for a user to specify a primitive argument value of `undef` this is still good programming practice.

such that the recipe is aborted and an error message printed. Note that this block runs in its own scope to prevent warnings from Perl concerning the masking of previous `OBEYW_STATUS` variables.

If automatic checking is not required, simply check the return status. If the parser finds an equals sign before the `obeyw` the line will not be re-written:

```
my $status = $Mon{'kappa_mon'}->obeyw("stats","ndf=$in");
```

- Line 54 retrieves a parameter value from the external task. This example also makes use of the automatic status checking within ORAC-DR. Whenever the parser sees the special `ORAC_STATUS` variable in a primitive, code is automatically added after this line to check the value of `ORAC_STATUS` and compare it with `ORAC__OK`. If the status is not good, the recipe aborts and an error message is printed. This saves the primitive writer from having to worry about status checking.
- Line 57 makes use of the `orac_print` command to send a message to the user. The `orac_print` command is written to send the message to multiple output filehandles as defined by the user with the `-log` switch to ORAC-DR.
- The final task in the loop (line 61) is to update the file name stored in the frame object. On exit from each primitive the frame and group objects must contain the filenames that should be used by subsequent primitives. This step is vital, and without it subsequent primitives will use the wrong input file names.
- The final step is to display the reduced frames. The `display_data` method will ask for the current frame to be displayed. Note that the display sub-system will *only* display the data frame if the user has requested this by configuring the display system (using, for example, the `oracdisp` command) accordingly. The check to make sure the display object is initialised is required in case the user has turned off the display system.

5.1 Log Files

It is sometimes desirable to write results to log files as data files are processed (for example, seeing statistics, pointing offsets etc). Rather than force the primitive writer to check for the existence of log files and decide whether or not to open or append to log files, the ORAC-DR system provides a simplified access to log file creation via the `ORAC:LogFile` class.

All that is required to write an entry to a log file is for the following methods to be invoked:

```
my $log = new ORAC::LogFile('log.whatever');
$log->header(@header);
$log->addentry(@lines);
```

The header will only be written to the log file if the log file does not previously exist so it is safe to run this command in a primitive without an explicit check. Both the `header()` and `addentry()` methods accept arrays, and newline characters will be appended to each item in the array when written to the log file. The convention is that all log file names should start with '`log.`'.

5.2 Temporary and intermediate files

In many cases, it is necessary to make use of temporary files within a primitive, either for intermediate data steps that are not relevant for the frame, or as text files input to external tasks. Since these are not required once the primitive is finished a class is provided for dealing with temporary files (`ORAC::TempFile`).

This class will choose a filename and, optionally, open the file ready for read-write access (when this facility is used it is guaranteed that the file is unique and will not overwrite any existing file). The file, and any files of the same name but with a `.sdf` extension[6], are removed when the variable goes out of scope.

The only files that should remain when a primitive completes should be those registered with the current frame or the current group. All others should be temporary and should be tidied up on leaving the primitive (which is automatic if `ORAC::TempFile` is used)[7]. This allows the final tidyup primitive to be responsible solely for removing unwanted intermediate frames that were the product of individual primitives (every time a the file name is updated in a frame object the previous value is stored for possible later removal by the tidy primitive).

5.3 Passing information between primitives

Since each primitive is evaluated in its own scope, it is not possible (or even desirable) to pass simple variables between separate primitives. Two means are provided for doing this:

- Using the `%_PRIMITIVE_NAME_` hash. The argument hash for each primitive is visible to all other primitives at the same level. This is because primitives are translated into:

  ```
  my %_PRIMITIVE_1_ = read_arguments( ... );
  {
    _PRIMITIVE_1_
  }
  my %_PRIMITIVE_2_ = read_arguments( ... );
  {
    _PRIMITIVE_2_
  }
  ```

  This works but has a number of problems:

  – The system breaks if the previous primitive is removed from the recipe.[8]

  – The system breaks if the primitive is renamed.

  – If the primitive from which data are required is in a different scope from the current primitive this will not work.

  – It feels too much like a global variable...

[6]the extension used for Starlink N-Dimensional data format (NDF)

[7]This is not always a good idea when debugging. Future versions of the pipeline will disable the removal of temporary files when the `-debug` flag is in use

[8]It will not simply return `undef`. The recipe will fail to run since the hash would not have been declared previously.

- Store the information with the current frame or group object. Both frame and group objects can use the `uhdr()` method to store arbitrary data (including references) in a hash. This is the recommended way of transferring data between primitives when the information relates to the current frame or group. By convention, the `hdr()` method should be used for storing FITS-like data. Checks should be made for the existence of data in the hash before using it.

The first method using the primitive hash only allows information to be passed within the current recipe whereas the frame header allows the information to be retained for subsequent frame processing.

6 Calibration

The calibration system is essentially based on the concept of index files. An index file is a data file containing information on all calibration observation reduced by the pipeline. A separate index file is created for each calibration sub-system (e.g. one for dark observations, one for skydip observations etc) with the convention is that each index file is stored in `ORAC_DATA_OUT` and prefixed with the string `index` (e.g. `index.dark`, `index.skydip` etc.). It is the responsibility of a primitive (usually a complete recipe is dedicated to the calibration observation) to file a calibration to an index file. The index file can be used simply to register a file name (e.g. the name of a dark file or flatfield) or a calibration result (the current sky opacity or flux conversion factor). Methods are provided in the `ORAC::Index` class for retrieving this information from the index file[9]

The index object is responsible for searching the relevant index file and returning the most suitable calibration. This is achieved by the use of external rules files (stored in `ORAC_DATA_CAL` called `rules.CALIBRATION_NAME`) which list which header keywords are relevant and should be checked against the headers of the current frame. This means that the calibration object itself does not need to worry about searching the index file or reading the rules files.

An example rules file could look like:

```
# Example rules file for SCUBA skydips
MODE eq 'SKYDIP'
FILTER eq $Hdr{FILTER}
WAVE == $Hdr{WAVE}
TAUZ
ORACTIME ; abs(ORACTIME - $Hdr{ORACTIME}) < 0.5
```

The format is intended to be fairly simple as it should be possible for a non-programmer to edit it. The main points are:

- # is the comment character. Anything after a comment within a line is ignored.
- Blank lines are ignored.

[9]For efficiency, the index file is kept in memory rather than read from disk every time it is to be accessed. This means that ORAC-DR is not guaranteed to work if two processes are sharing a single ouput data directory as this would cause problems with index file updates [they are not locked].

- Each line in the rules file is translated to the Perl code and evaluated. If it returns 'true' the value in the index file (keyed by the first word in the rules file) matches the coresponding values in the header of the current frame. For example, the second rule is translated into the following code (assuming the current index entry contains the filter name of "850W").

```
if ( '850W' eq $Hdr{FILTER} ) {
   # Rules passes
}
```

  and the last line in the above rules file could be translated to the following `if` statement (using ORACTIME of 19990827.55 for the index entry):

```
'19990827.55'; abs(19990827.55 - $Hdr{ORACTIME}) < 0.5
```

  which returns true if the `ORACTIME` stored in the index is within half a day of the value stored in the current frame object (%Hdr is a hash read from the current frame header). The semi-colon is used to separate the Perl code into two statements. The return value of the `eval` is that of the second statement. This is required for cases that are not simply '`A == B`' format.

- When using complex rules the substituted index value (ORACTIME in the above example) is only quoted when placed as the first statement. When in the body of the rule it must be quoted if the value is not a number. This is because the value itself is placed into the rule rather than a variable containing the value.

- Lines which only contain a keyword, are place holders indicating that the information should be present in the index file but not used. Care must be taken that the value of the corresponding keyword can never be zero since that would return false to the rules system. If this is the case a rule of

```
KEYWORD ; 1
```

  can be used instead since in this case the test will always return true.

6.1 Calibration overrides

ORAC-DR gives the user the ability to override the default calibration information as returned by the pipeline through use of the `-calib` commandline option. By adding methods in `ORAC::Calib` or in instrument-specific subclasses, it is possible to add various override methods. As an example of this, see the `profile`, `profilename`, `profileindex`, and `profilenoupdate` methods in `ORAC::Calib::CGS4`. For further information on adding override methods see §9.3, and for information on using override methods when reducing data with ORAC-DR see SUN/230.

A number of general overrides are available.

- baseshift - Use the given comma separated doublet (i.e. "0,0") as the frame's base position.
- bias - Use the given bias frame.
- dark - Use the given dark frame.
- flat - Use the given flat frame.

- mask - Use the given mask. Usually used for bad pixel masks.
- readnoise - Use the given value for the detector readnoise.
- rotation - Use the given frame as a rotation matrix.
- sky - Use the given sky frame.
- standard - Use the given standard star frame.

7 Message output

The ORAC-DR infrastructure provides commands for primitive writers to print informational messages, warnings and error messages to the user. These commands are `orac_print`, `orac_say`, `orac_warn`, and `orac_err`. In principal these commands can send information to different locations and this location is under the control of the person running the pipeline rather than primitive writer (using the `-log` option). The Perl `print` and `warn` commands should not be used since they always write output to standard output and standard error whereas the ORAC print commands can send to multiple filehandles.

8 Recipe Debugging

Some facilities are provided to ease the debugging of recipes and primitives. These are as follows.

- When a recipe will not compile (due to syntax errors, for example) the pipeline aborts and the error message and a listing of relevant recipe lines (the version of the recipe executed internally with all the extra code inserted) is printed to the screen.
- The `-w` switch should be used during development in order to trap and fix any warnings detected by the Perl interpreter at runtime. In many cases, these warnings are indicative of code that makes assumptions about state (warnings on use of `undef`) or scope (`my` declarations masking variables defined in a higher scope).

  If warnings are raised from the ORAC-DR core please contact the authors and they will be fixed.[10]
- The `-verbose` switch can be used to turn on messages from the algorithm engines. This is sometimes useful to make sure that the external task is doing the expected operation.
- The `oracdr_parse_recipe` command is provided to translate a recipe into Perl code. It takes a recipe name as argument and sends the translated recipe to standard output. This command can be used to check the syntax of the recipe (by compiling the perl code and exiting immediately) by using the `-syntax` option.

[10]currently the one known issue is multiple declarations of the `ORAC_PRIMITIVE` variable.

- The final option is to turn on full debugging with the `-debug` switch. This creates a file in `ORAC_DATA_OUT` called `ORACDR.DEBUG` containing the contents of each message sent to external tasks with an `obeyw`. This can be used to find out what the final message sent to a task was before the recipe crashed. If a recipe crashes when the debugging is turned on the full contents of the recipe buffer (i.e. the fully parsed recipe) will also be written to a file in `ORAC_DATA_OUT` called `ORACDR_RECIPE.dump`. This would be identical to running the `oracdr_parse_recipe` command.

The Perl debugger should be used with care since it is possible to hang the message bus if the program is frozen during a message transaction and the debugger is not optimized for use with the use of nested `eval` common in the pipeline internals.

9 Adding new instruments

Adding a new instrument to ORAC-DR requires a number of steps, the complexity of which will depend on how close the instrument is to an instrument that is already supported by the pipeline.

This section describes the areas that must be modified to support a new instrument. It assumes knowledge of the following Perl concepts: writing object-oriented modules, lexical variables, Perl data structures and `eval`. Message system interfaces will also require a knowledge of Perl XS.

9.1 Frames

An `ORAC::Frame` class is responsible for filename conventions and handling of sub-frames. The most common methods that need to be sub-classed are:

new()

The object constructor should be modified to set the default behaviour for `rawfixedpart()`, (the part of the filename that does not change with UT date of observation number), `rawsuffix()`, (the file suffix of the raw data file), `rawformat()` (the input format of the data: FITS, NDF etc.) and `format()` (the data format required by the algorithm engines).

The base constructor (`SUPER::new()`) should be invoked to instantiate the object.

calc_orac_headers()

This method is used to translate instrument specific headers to those required by the pipeline. Currently this method must insert the following keywords into the frame header.

ORACUT

This should be the UT date of the observation in YYYYMMDD format.

ORACTIME

This should be the UT date and time of the observation in YYYYMMDD.frac format (i.e. UT date plus fraction of day). This is used by the calibration system to ensure that a guaranteed time stamp can be used for comparison in index files.

file_from_bits()

Given a UT date and an observation number, return the name of the raw data file. This is used by the pipeline to look for the file on disk.

flag_from_bits()

Given a UT date and an observation number, return the name of the flag file. This is used by the pipeline to search for a completion flag on disk associated with the current observation file. Only required if it is intended for the pipeline to be used in conjunction with the `-loop flag` option.

findgroup()

Translates the header values into a group name and updates the frame.

findrecipe()

Translates the header values into a recipe name and updates the frame.

template()

Modifies the current filename so that it matches the filename that you would have had at an earlier point in the reduction when the supplied template was valid. This is used for jumping to earlier steps in the reduction when performing group operations. Currently this method is generally implemented by replacing the suffix with the supplied value. It could also be implemented by accessing the list of prior file names with the `intermediates()` method.

inout()

Given a suffix and sub-frame number provide a recommended name for the output file. Does not update the frame object. In general, this can either simply append the suffix or replace the last suffix. The latter is normal practice.

In addition to those listed above, some methods need to know something about the file format and should be modified (most UKIRT and JCMT instruments inherit from the `ORAC::Frame::NDF` class rather than `ORAC::Frame` since that class knows how to read from NDF files). The automatic format conversion will have occurred before these methods need to be run (for example if the input format is FITS but the processing format is NDF, then the methods should be written assuming NDF files). These methods are as follows.

readhdr()

Reads the file header and stores the resulting hash reference in the object. Also forces the ORAC headers to be calculated.

erase()

A method to delete the file currently associated with the frame. This is effectively and `unlink()` but for NDF files the `.sdf` is appended. The base class implementation can be used in most cases.

file_exists()

Checks to see if the data file exists. There is a special case for NDF format since the `.sdf` extension is not present in the file name.

stripfname()

Strips the extension from the filename. The base class does nothing to the filename.

Finally, any extra methods not present in the base class but required by an instrument should also be added. These can cover translation of wavelengths to sub-frame filename, for example (used by the SCUBA class).

9.1.1 Frame headers

There are a number of internal headers that ORAC primitives rely on to correctly reduce data. These headers are retrieved from the files and translated to be used by oracdr. These headers can be used in recipes through the `ORAC::Frame::uhdr()` method and prepending ORAC_ to each header. For example, to refer to the `INSTRUMENT` header in a primitive, you would use

```
my $Frm = new ORAC::Frame('filename.sdf');
my $instrument = $Frm->uhdr('ORAC_INSTRUMENT');
```

New internal headers can always be created, if necessary, to define some ancillary data needed by a recipe, in an instrument-independent fashion.

The following headers currently exist. Note that not all headers are defined for each instrument or recipe.

AIRMASS_START

Airmass at start of the observation (for a mosaic) or integration (for a raw frame).

AIRMASS_END

Airmass at end of the observation (for a mosaic) or integration (for a raw frame).

CHOP_ANGLE

Position angle of the chop (in degrees).

CHOP_THROW

Throw of the chop with respect to the middle position (in arcseconds).

CONFIGURATION_INDEX

Configuration index. Increments when an instrument's hardware configuration changes.

DEC_BASE

Declination (ORAC_EQUINOX equinox) at the reference position at zero offset (in degrees). The reference position is normally at the array centre, but may be displaced, for example, to avoid the intersections of abutted detector sub-arrays.

DEC_SCALE

Pixel increment along declination axis (in arcsec).

DEC_TELESCOPE_OFFSET

Telescope declination offset with respect to the base position given by DEC_BASE (in arcseconds).

DETECTOR_BIAS

Detector bias voltage (in volts)

DETECTOR_INDEX

Position number in detector scan.

DETECTOR_READ_TYPE

Observing mode, such as `STARE`, `NDSTARE`, `CHOP`.

EQUINOX

Equinox of object position (in years). It should be `2000`.

EXPOSURE_TIME

Integration time per exposure (in seconds).

FILTER

Combined filter name.

GAIN

Detector gain (in electrons/ADU).

GRATING_DISPERSION

Grating dispersion (in microns/pixel).

GRATING_NAME

Grating name.

GRATING_ORDER

Grating order.

GRATING_WAVELENGTH

Grating wavelength (in microns).

INSTRUMENT

Instrument name.

NSCAN_POSITIONS

Number of scan positions in scan.

NUMBER_OF_EXPOSURES

Number of exposures in integration.

NUMBER_OF_OFFSETS

Number of jitter offset positions in an observation.

NUMBER_OF_READS

Number of reads per exposure.

OBJECT

Object name from telescope.

OBSERVATION_MODE

Camera mode, such as `imaging` or `spectroscopy`. Used for multi-mode instruments like Michelle.

OBSERVATION_NUMBER

Observation number. Observation numbers normally commence at `1` for each night.

OBSERVATION_TYPE

Observation type. Used to determine whether observation is of the `OBJECT`, `DARK`, or ARC, for example.

POLARIMETRY

Whether or not polarimetry observations are being done. True (1) if so, false (0) otherwise.

RA_BASE

Right Ascension (ORAC_EQUINOX equinox) at the reference position at zero offset (in degrees). See DEC_SCALE.

RA_SCALE

Pixel increment along right-ascension axis (in arcsec).

RA_TELESCOPE_OFFSET

Telescope right-ascension offset with respect to the base position given by RA_BASE (in arcseconds).

ROTATION

Angle of declination axis with respect to the second (y) axis, measured counterclockwise (in degrees).

SCAN_INCREMENT

Increment between scan positions (in pixels).

SLIT_ANGLE

Position angle of slit (in degrees)

SLIT_NAME

Name of slit.

SPEED_GAIN

Readout speed. The default is `Normal`.

STANDARD

Is the target a standard-star observation?

UTDATE

UTC date when the observation was made.

UTEND

End time of integration (for raw data) or the observation (for a mosaic).

UTSTART

Start time of integration (for raw data) or the observation (for a mosaic).

WAVEPLATE_ANGLE

Polarimeter waveplate position angle (in degrees).

X_DIM

Number of detectors in a readout column (in pixels).

Y_DIM

Number of detectors in a readout row (in pixels).

X_LOWER_BOUND

Start column of array readout.

X_UPPER_BOUND

End column of array readout.

Y_LOWER_BOUND

Start row of array readout.

Y_UPPER_BOUND

End row of array readout.

9.2 Groups

An `ORAC::Group` class is responsible for dealing with groups of frame objects and with an output group data file. The most common methods that need to be sub-classed are given below.

new()

The constructor must be overridden to specify the type fixed part and file suffix of the output group file. The base constructor should be used from the subclass to instantiate the object.

readhdr()/erase()/file_exists()/file_from_bits()

Similar to the frame implementations.

For specialized applications it may also be necessary to implement the `coaddswrite()`/`coaddsread()` methods. These will be used by the CGS4/MICHELLE instrument classes to allow coadds to be combined part way through a group without having to re-reduce earlier observations. Not all instruments require support for this.

9.3 Calibration

For a new instrument all that is required is to specify a new set of rules and work out what to do with the 'best' calibration that is returned. For UFTI this simply means that the name is returned to the primitive for use. For SCUBA more complex routines are required for sky opacity calibration since the number returned by the index file is sometimes modified for different filters before being returned to the user (the SCUBA calibration system never returns a file name).

For every type of calibration, the calibration object (`ORAC::Calib` and sub-classes) must provide the following methods:

(1) A method for returning the current calibration observation that matches the criteria. This method should be an obvious name related directly to the calibration (e.g. 'dark', 'skydip').

(2) A method for retrieving the `ORAC::Index` object associated with the relevant index file. The name of this method should be related to the name of the index file (e.g. `skydipindex` will access the file `index.skydip`).

(3) A method to prevent the pipeline from overwriting the current calibration (used when the user is overriding a certain calibration from the command line). This will be called, for example, `darknoupdate()`. ORAC-DR assumes that the method is of this form since this method will be called whenever a user makes use of the `-calib` switch,

9.4 Algorithm Engines

Each instrument requires its own implementation of an `ORAC::Inst` module. This module is responsible for implementing the commands for starting the messaging layer (if required) and launching external tasks ready for use by the pipeline. It may be possible for future releases to launch external tasks on demand the first time an `obeyw` is issued but this depends on the implementation of the messaging class and still requires that an object for each class is instantiated in the `ORAC::Inst` module at startup.

Currently, this module is not object-oriented. This may change in the future.

Another factor related to the algorithm engines is the implementation of the messaging interface. The current interface is to the ADAM messaging system used by Starlink. An additional interface is provided to run ADAM tasks via the Unix shell as a proof of concept. Interfaces to Glish and IRAF will be needed to talk to AIPS++ and IRAF tasks. A Perl-to-DRAMA interface exists and it should be fairly simple to write an ORAC-DR interface for DRAMA tasks. A shell interface should be used as a last resort and only if valid exit status can be returned to the pipeline.

9.5 ORAC-DR

Once the instrument classes have been written, ORAC-DR needs to be modified so that it can interpret a new value for `ORAC_INSTRUMENT` and configure the frame, group and calibration objects correctly.

In the future this configuration may be moved into an external data file so that the core ORAC-DR routine need not be modified by people porting the system to a new instrument.

9.6 Recipes

Once ORAC-DR has been configured to recognise the new instrument, the final step is to write the recipes and associated primitives to deal with the new data and possibly new algorithm engines. At its simplest, for example for an infrared image, this may simply require copying pre-written primitives from an existing instrument[11]. At its most complex, completely new primitives will have to be written.

A Directory Layout

ORAC-DR is designed to include all its directories in a single location and not to care where that location is. All ORAC-DR software is located relative to the directory described by the `ORAC_DIR` environment variable.

The standard layout is as follows.

bin

Executable programs. This includes ORAC-DR, `oracman` and others. Note that none of the programs that form part of ORAC-DR are binary. The core ORAC-DR system will run anywhere the necessary Perl modules are available.

recipes

All the recipe files. Contains a directory for each instrument (using upper case) usually matching the value of the `ORAC_INSTRUMENT` environment variable.

primitives

All the primitive files. Contains a directory for each instrument (using upper case) usually matching the value of the `ORAC_INSTRUMENT` environment variable.

howto

Basic introductory documentation for ORAC-DR.

lib

Library files required by ORAC-DR. Contains a `perl5` directory with all the Perl modules used by the system. The `perl5` directory is the usual value for the `ORAC_PERL5LIB` environment variable.

images

Image files used by the system. Contains the start up images for the display tools (in NDF format).

docs

Main documentation. Contains a directory per document. Currently contains all the Starlink User Notes.

[11] Plans are ongoing to allow multiple search paths for recipes and primitives so that single copies can be used by multiple instruments without having to copy the files to separate locations

gui

Graphical user interface definitions. Only used by `oracdisp`.

B Perl Bibliography

The following Perl books are recommended for more information on the Perl features described in this document:

Programming Perl 3rd edition, Wall, Christiansen and Orwant, 2000, O'Reilly and Associates.

Advanced Perl Programming Srinivasan, 1997, O'Reilly and Associates.

Object-Oriented Programming in Perl Conway, 1999, Manning

C Class libraries

This section describes the class libraries that are relevant for a recipe writer.

C.1 ORAC::Bounds

Provide spatial and/or spectral bounds for files.

SYNOPSIS

```
use ORAC::Bounds qw/ retrieve_bounds update_bounds_headers /;

my $bounds = retrieve_bounds( $filename );
update_headers( $filename );
```

DESCRIPTION

This package provides functions to retrieve spatial and spectral information about an NDF containing AST FrameSet information. It currently retrieves the corners of a bounding box in the spatial extent, and the upper and lower bounds for the frequency extent.

Bounding information will be incorrect for data encompassing a pole.

METHODS

retrieve_bounds

This method retrieve spatial/spectral bounds for a given NDF.

```
my $bounds = retrieve_bounds( $file );
```

The file must be an NDF. The filename need not have the '.sdf' extension.

The return value is a hash reference containing the following keys:

reference
Astro::Coords object for the reference sky position.

top_left
Astro::Coords object for the top left corner of the bounding box.

top_right
Astro::Coords object for the top right corner of the bounding box.

bottom_left
Astro::Coords object for the bottom left corner of the bounding box.

bottom_right
Astro::Coords object for the bottom right corner of the bounding box.

centre
Astro::Coords object for the central pixel in the NDF.

frq_sig_lo
Barycentric frequency, in gigahertz, of the lower end of the signal sideband.

frq_sig_hi
Barycentric frequency, in gigahertz, of the upper end of the signal sideband.

frq_img_lo
Barycentric frequency, in gigahertz, of the lower end of the image sideband.

frq_img_hi
Barycentric frequency, in gigahertz, of the upper end of the image sideband.

A specific key will be undefined if the given file does not have the appropriate information to calculate it with. For example, if a file does not have a SkyFrame, then none of the Astro::Coords objects listed above will be defined.

This function currently only works on files that have 3D CmpFrames.

return_bounds_header

This function creates an `Astro::FITS::Header` object which describes the bounds of the observation in question.

```
  $header = return_bounds_header( $file );
```

The file must be an NDF. The '.sdf' extension need not be present.

The following mappings from the keys listed in the retrieve_bounds() function are made:

reference -> OBSRA, OBSDEC

bottom_left -> OBSRABL, OBSDECBL

bottom_right -> OBSRABR, OBSDECBR

top_left -> OBSRATL, OBSDECTL

top_right -> OBSRATR, OBSDECTR

freq_img_lo -> FRQIMGLO

freq_img_hi -> FRQIMGHI

freq_sig_lo -> FRQSIGLO

freq_sig_hi -> FRQSIGHI

If any of the headers already exist, they will be overwritten.

update_bounds_headers

This function sets FITS headers in the file describing the bounds of the observation in question.

```
  update_bounds_headers( $file );
```

The file must be an NDF. The '.sdf' extension need not be present.

The following mappings from the keys listed in the retrieve_bounds() function are made:

reference -> OBSRA, OBSDEC

bottom_left -> OBSRABL, OBSDECBL

bottom_right -> OBSRABR, OBSDECBR

top_left -> OBSRATL, OBSDECTL

top_right -> **OBSRATR, OBSDECTR**

freq_img_lo -> **FRQIMGLO**

freq_img_hi -> **FRQIMGHI**

freq_sig_lo -> **FRQSIGLO**

freq_sig_hi -> **FRQSIGHI**

If any of the output keys from retrieve_bounds() are undefined, then the corresponding headers will not be written to the file.

SEE ALSO

Starlink::AST, Astro::Coords.

COPYRIGHT

Copyright (C) 2007 Science and Technology Facilities Council. All Rights Reserved.

C.2 ORAC::Calib

Base class for selecting calibration frames in ORAC-DR

SYNOPSIS

```
use ORAC::Calib;

$Cal = new ORAC::Calib;

$dark = $Cal->dark;
$Cal->dark("darkname");

$Cal->standard(undef);
$standard = $Cal->standard;
$bias = $Cal->bias;
```

DESCRIPTION

This module provides the basic methods available to all ORAC::Calib objects. This class should be used for selecting calibration frames.

Unless specified otherwise, a calibration frame is selected by first, the nearest reduced frame; second, explicit specification via the -calib command line option (handled by the pipeline); third, by search of the appropriate index file.

Note this version: Index files not implemented.

PUBLIC METHODS

The following methods are available in this class.

Constructors

new

Create a new instance of a ORAC::Calib object. The object identifier is returned.

```
$Cal = new ORAC::Calib;
```

Accessor Methods

thing

Returns the hash that can be used for checking the validity of calibration frames. This is a combination of the two hashes stored in `thingone` and `thingtwo`. The hash returned by this method is readonly.

```
$hdr = $Cal->thing;
```

thingone

Returns or sets the hash associated with the header of the object (frame or group or whatever) needed to match calibration criteria against.

Ending sentences with a preposition is a bug.

thingtwo

Returns or sets the hash associated with the user defined header of the object (frame or group or whatever) against which calibration criteria are applied.

General Methods

find_file

Returns the full path and filename of the requested file (the first file found in the search path).

```
$filename = $Cal->find_file("fs_izjhklm.dat");
```

croaks if the file can not be found. It's likely that this is a bit drastic but it will indicate something bad is going on before some other unexpected behaviour occurs. See **ORAC::Inst::Defn::orac_determine_calibration_search_path** for information on setting up calibration directories.

retrieve_by_column

Returns the value for the specified column in the specified index.

```
$value = $Cal->retrieve_by_column( "readnoise", "ORACTIME" );
```

The first argument is a queryable

DYNAMIC METHODS

These methods create methods for the standard calibration schemes for subclasses. By default calibration "xxx" needs to create standard accessors for "xxxnoupdate", "xxxname" and "xxxindex".

GenericIndex

Helper routine that creates an index object and returns it. Updates the object based on the root name.

```
 $index = $Cal->CreateIndex( "flat", "dynamic" );
```

Where the first argument should match the root name of the index and rules file. The second argument can have three modes:

```
  dynamic - index file is assumed to be in ORAC_DATA_OUT
  static  - index file is assumed to be in the calibration tree
  copy    - index file will be copied to ORAC_DATA_OUT from
            ORAC_DATA_CAL if not present in ORAC_DATA_OUT
```

If a third argument is supplied it is assumed to be an ORAC::Index object to be stored in the calibration object.

GenericIndexAccessor

Generic method for retrieving or setting the current value based on index and verification. Uses ORACTIME to verify.

```
  $val = $Cal->GenericIndexAccessor( "sky", 0, 1, 0, 1, @_ );
```

First argument indicates the root name for methods to be called. ie "sky" would call "skyname", "skynoupdate", and "skyindex".

Second argument controls whether the time comparison should be nearest in time (0), or earlier in time (-1).

Third argument controls croaking behaviour. False indicates that the method should croak if a suitable calibration can not be found. True indicates that it should return undef. If a code ref is provided, it will be executed if no suitable calibration is found. If it returns a defined value it will be assumed to be a valid match, and if it returns undef the method will croak as no suitable calibration will be available. This allows defaults to be inserted.

Fourth argument controls whether calibration object verification is not done.

Fifth argument controls whether warnings are displayed when searching through the index file for a suitable calibration. Default is to warn.

```
  $val = $Cal->GenericIndexAccessor( "mask", 0, sub { return "bpm.sdf" }, @_ );
```

GenericIndexEntryAccessor

Like `GenericIndexAccessor` except that a particular value from the index is retrieved rather than a indexing key (filename).

No verification is performed.

```
  $val = $Cal->GenericIndexAccessor( "sky", "INDEX_COLUMN", @_ );
```

First argument indicates the root name for methods to be called. ie "sky" would call "skycache", "skynoupdate", and "skyindex".

If a reference to an array or columns is given in argument 2, all values are checked and the row reference is returned instead of a single value.

```
  $entryref = $Cal->GenericIndexAccessor( "sky", [qw/ col1 col2 /], @_ );
```

CreateBasicAccessors

Dynamically create default accessors for "xxxnoupdate", "xxxname" and "xxxindex" methods.

```
 __PACKAGE__->CreateAccessors( "xxx", "yyy", "zzz" );
```

SEE ALSO

ORAC::Group and *ORAC::Frame*

COPYRIGHT

C.3 ORAC::Calib::WFCAM;

SYNOPSIS

use ORAC::Calib::WFCAM;

```
  $Cal = new ORAC::Calib::WFCAM;

  $dark = $Cal->dark;
  $Cal->dark("darkname");
```

DESCRIPTION

This module contains methods for specifying WFCAM-specific calibration objects when using Starlink software for reduction. It provides a class derived from ORAC::Calib::Imaging. All the methods available to ORAC::Calib::Imaging objects are available to ORAC::Calib::WFCAM objects.

METHODS

The following methods are available:

General Methods

interleavemask

Determine the mask necessary for microstep interleaving.

```
  $interleavemask = $Cal->interleavemask;
```

This method returns a filename, including directory structure. If the noupdate flag is set there is no verification that the mask meets the specified rules.

dark

Return (or set) the name of the current dark - checks suitability on return. This is subclassed for WFCAM so that the warning messages when going through the list of possible darks are suppressed.

flat

Return (or set) the name of the current flat.

```
  $flat = $Cal->flat;
```

This method is subclassed for WFCAM so that the warning messages when going through the list of possible flats are suppressed.

mask

Return (or set) the name of the current bad pixel mask.

```
  $mask = $Cal->mask;
```

This method is subclassed for WFCAM because we have one mask per camera and not one standard mask.

skyflat

Return (or set) the name of the current skyflat.

```
  $skyflat = $Cal->skyflat;
```

Support Methods Each of the methods above has a support implementation to obtain the index file, current name and whether the value can be updated or not. For method "cal" there will be corresponding methods "calindex", "calname" and "calnoupdate". "calcache" is an allowed synonym for "calname".

```
  $current = $Cal->calcache();
  $index = $Cal->calindex();
  $noup = $Cal->calnoupdate();
```

Additionally, "flat" and "mask" are locally modified to support a static index location.

COPYRIGHT

C.4 ORAC::Calib::ACSIS;

SYNOPSIS

```
  use ORAC::Calib::ACSIS;

  $Cal = new ORAC::Calib::ACSIS;
```

DESCRIPTION

This module contains methods for specifying ACSIS-specific calibration objects. It provides a class derived from ORAC::Calib. All the methods available to ORAC::Calib objects are also available to ORAC::Calib::ACSIS objects.

METHODS

The following methods are available:

Constructor

new

Sub-classed constructor. Adds knowledge of pointing, reference spectrum, beam efficiency, and other ACSIS-specific calibration information.

Accessors

bad_receptors

Set or retrieve the name of the system to be used for bad receptor determination. Allowed values are:

- master
  Use the master index.bad_receptors index file in $ORAC_DATA_CAL.
- index
  Use the index.bad_receptors_qa index file in $ORAC_DATA_OUT as generated by the pipeline.
- indexormaster
  Use both the master index.bad_receptors and pipeline-generated index.bad_receptors_qa file. Results are 'or'ed together, so any receptors flagged as bad in either index file will be flagged as bad.
- file
  Use the contents of the file *bad_receptors.lis*, which contains a space-separated list of receptor names in the first line. This file must be found in $ORAC_DATA_OUT. If the file cannot be found, no receptors will be flagged.

- 'list'
  A colon-separated list of receptor names can be supplied. This list can be in combination with one of the other options.

The default is to use the 'indexormaster' method. The returned value will always be in upper-case.

bad_receptorsindex

Return (or set) the index object associated with the master bad receptors index file. This index file is used if bad_receptors() is set to 'MASTER' or 'INDEXORMASTER'.

bad_receptors_qa_index

Return (or set) the index object associated with the pipeline-generated bad receptors index file. This index file is used if bad_receptors() is set to 'INDEX' or 'INDEXORMASTER'.

bad_receptors_list

Return a list of receptor names that should be masked as bad for the current observation. The source of this list depends on the setting of the bad_receptors() accessor.

flat

Retrieve flat-field ratios for the observation's UT date.

standard

Retrieve the relevant standard.

receptor_names

This returns the permitted receptors names for the current instrument. It first looks for the header, and failing that supplies the default set.

```
@receptors = $Cal->receptor_names();
@receptors = $Cal->receptor_names($instrument);
```

sidebandcorr_factor

Calculate the sideband correction factor. Requires an instrument, a time and an lofrequency, which it gets from the headers by default

```
$factor = $Cal->sidebandcorr_factor();
```

Optionally you can specify the instrument, ut and lo frequency (GHz) in the call.

```
$factor = $Cal->sidebandcorr_factor($instrument, $ut, $lo_freq);
```

Support Methods Each of the methods above has a support implementation to obtain the index file, current name and whether the value can be updated or not. For method "cal" there will be corresponding methods "calindex", "calname" and "calnoupdate". "calcache" is an allowed synonym for "calname".

```
$current = $Cal->calcache();
$index = $Cal->calindex();
$noup = $Cal->calnoupdate();
```

COPYRIGHT

C.5 ORAC::Calib::SCUBA

SCUBA calibration object

SYNOPSIS

```
use ORAC::Calib::SCUBA;

$Cal = new ORAC::Calib::SCUBA;

$gain = $Cal->gain($filter);
$tau  = $Cal->tau($filter);
@badbols = $Cal->badbols;
```

DESCRIPTION

This module returns (and can be used to set) calibration information for SCUBA. SCUBA calibrations are used for extinction correction (the sky opacity) and conversion of volts to Janskys.

It can also be used to set and retrieve lists of bad bolometers generated by noise observations.

This class does inherit from **ORAC::Calib** although nearly all the methods in the base class are irrelevant to SCUBA (this class only uses the thing() method).

PUBLIC METHODS

The following methods are available in this class. These are in addition to the methods inherited from **ORAC::Calib**.

Constructor

new

Create a new instance of a ORAC::Calib::SCUBA object. The object identifier is returned.

```
$Cal = new ORAC::Calib::SCUBA;
```

Accessor Methods

default_fcfs

Return the default FCF lookup table indexed by filte.

```
 %FCFS = $cal->default_fcfs();
```

secondary_calibrator_fluxes

Return the lookup table of fluxes for secondary calibrators.

```
 %photfluxes = $cal->secondary_calibrator_fluxes();
```

badbols

Set or retrieve the name of the system to be used for bad bolometer determination. Allowed values are:

- index

  Use an index file generated by noise observations using the reflector blade. The bolometers stored in this file are those that were above the noise threshold in the _REDUCE_NOISE_ primitive. The index file is generated by the _REDUCE_NOISE_ primitive
- file

  Uses the contents of the file *badbol.lis* (contains a space separated list of bolometer names in the first line). This file is in ORAC_DATA_OUT. If the file is not found, no bolometers will be flagged.
- 'list'

  A colon-separated list of bolometer names can be supplied. If badbols=h7:i12:g4,... then this list will be used as the bad bolometers throughout the reduction.

Default is to use the 'file' method. The value is always upper-cased.

General methods

badbol_list

Returns list of bolometer names that should be turned off for the current observation. The source of this list depends on the setting of the badbols() parameter (controlled by the user). Can be one of 'index', 'file' or actual bolometer list. See the badbols() method documentation for more information.

SEE ALSO

ORAC::Calib::JCMTCont

COPYRIGHT

C.6 ORAC::Calib::SCUBA2

SCUBA-2 calibration object

SYNOPSIS

```
use ORAC::Calib::SCUBA2;

$Cal = new ORAC::Calib::SCUBA2;
```

DESCRIPTION

This module returns (and can be used to set) calibration information for SCUBA-2.

It can also be used to set and retrieve lists of bad bolometers generated by noise observations.

This class does inherit from **ORAC::Calib** although nearly all the methods in the base class are irrelevant to SCUBA-2 (this class only uses the thing() method).

Note that currently this module is mostly just a copy of the SCUBA module, with extra SCUBA-2 specific methods. Some pruning/updating WILL be necessary.

PUBLIC METHODS

The following methods are available in this class. These are in addition to the methods inherited from **ORAC::Calib**.

Constructor

new

Create a new instance of a ORAC::Calib::SCUBA2 object. The object identifier is returned.

```
$Cal = new ORAC::Calib::SCUBA2;
```

Accessor Methods

beamcomp

Return the number of components in the current fit to the beam. Will be either 1 or 2 if a fit exists, otherwise undef.

```
my $ncomp = $Cal->beamcomp();
```

beamfit

A method to set or retrieve the full parameter set for the most recent fit to the beam. If setting the beam parameters, all of the parameters must be specified as a hash. The beam dimensions and orientation must be passed as array references. A hash reference containing the beam parameters is returned.

```
$Cal->beamfit( majfwhm => \@majfwhm, minfwhm => \@minfwhm,
               orient => \@orient, gamma => $gamma );

$Cal->beamfit( %beamfit );

my $beamfit_ref = $Cal->beamfit;
```

The returned hash reference has the following keys: `BeamA`, `BeamAErr`, `BeamB`, `BeamBErr`, `PA`, `PAErr`, `FWHM`, `Gamma` and `BeamComp`. `BeamA` and `BeamB` are the major and minor axes respectively of the first component of the fit. In the case of a two-component fit, the FWHM of the second component (and its uncertainty) is stored by the `errbeam` method. The `FWHM` entry contains the geometric mean of the major and minor axes of the first component. `Gamma` will always be 2 if there are two components.

Conventionally the units of the FWHM are arcsec, but it is up to the caller to ensure that the FWHM values are in the units of choice before storing them here.

catalog_position

Get the catalog position for a secondary calibrator if we have a catalog position for it.

```
  my $position = $Cal->catalog_position($Frm->hdr('OBJECT'));
```

Returns undef if there is no catalog position stored for the given object. Otherwise returns a reference to a RA, Dec array of sexagesimal strings.

errbeam

The current estimated FWHM of the error beam (and its uncertainty) as given in the `beampar` results. Must be given and returns a hash reference with the keys `BeamA` and `BeamAErr`.

```
$Cal->errbeam({BeamA => $fwhm, BeamAErr => $fwhm_err});
my $errbeam = $Cal->errbeam;
```

errfrac

Fractional power in the error beam as determined from aperture photometry.

```
$Cal->errfrac($errfrac);
my $errfrac = $Cal->errfrac;
```

fwhm_err

Returns the FWHM of the current estimate of the error beam.

```
$Cal->fwhm_err($fwhm_err);
my $fwhm_err = $Cal->fwhm_err;
```

fwhm_fit

Retrieve the fitted beam full-width-at-half-maximum (FWHM). Returns the geometric mean of the major/minor axes of the fit.

```
my $fitted_fwhm = $Cal->fwhm_fit;
$Cal->fwhm_fit($fwhm);
```

If the complete beam parameters are required, the **beamfit** method should be used instead. Returns undef if no fit parameters have been stored.

Instance methods

beam

Return the telescope beam parameters for the current wavelength. See the `beamfit` method for the results of the most recent fit to the beam.

Returns a hash reference with the keys `FWHM1`, `FWHM2`, `AMP1`, `AMP2`, `FRAC1` and `FRAC2`.

```
my $telescope_beam = $Cal->beam;
```

beamamps

Return the relative amplitudes of the telescope beam components at the current wavelength. Returns either an array with two elements or array reference depending on caller. The first element corresponds to the main beam component.

```
my $beamamps = $Cal->beamamps;
my @beamamps = $Cal->beamamps;
```

beamarea

Returns the beam area in units of arcsec^2/beam. The nominal values have been determined empirically from an ensemble of calibration data.

```
$beamarea = $Cal->beamarea();
```

The optional parameter is an aperture diameter in arcsec:

```
  $beamarea = $Cal->beamarea( $diam );
```

This value can be thought of as an ideal Gaussian of FWHM sqrt(beamarea/1.133). It will be slightly bigger than the nominal FWHM of the primary beam because error lobes are included.

beamfrac

Return the fractional power in each component of the beam as an array or array reference. May also take parameter to return either the main or error beam fraction.

```
  my ($main, $err) = $Cal->beamfrac;
  my $err = $Cal->beamfrac("err");
```

default_fcfs

Return the default FCF lookup table indexed by filter.

```
 %FCFS = $cal->default_fcfs();
```

fwhm

Return the measured telescope beam FWHM for the two components at the current wavelength. Returns either an array with two elements or array reference depending on caller. The first element is the main beam component.

```
  my $fwhm = $Cal->fwhm;
  my @fwhm = $Cal->fwhm;
```

fwhm_eff

Returns the FWHM (in arcsec) of a Gaussian with the same area as the empirical telescope beam (see the **beamarea** method below).

```
  $fwhm_eff = $Cal->fwhm_eff;
```

nep_spec

Method to return the NEP spec for the current wavelength

```
  my $nep_spec = $Cal->nep_spec;
```

secondary_calibrator_fluxes

Return the lookup table of fluxes for secondary calibrators. Takes an optional parameter which returns the total fluxes rather than the 'per beam' values.

```
 %photfluxes = $cal->secondary_calibrator_fluxes( $ismap );
```

Index Methods

mask

Return (or set) the name of the current bad bolometer mask.

```
$mask = $Cal->mask;
```

This method is subclassed for SCUBA-2 because we have one mask per subarray and not one standard mask. Note that the user must set the subarray with the Frame class `subarray()` method before a suitable calibration entry can be found. This is due to the fact that it is not possible to search the Frame subheaders when evaluating the rules.

dark

Return (or set) the name of the current dark frame.

```
$dark = $Cal->dark;
```

This method is subclassed for SCUBA-2 because we have dark frames for each subarray. Note that the user must set the subarray with the Frame class `subarray()` method before a suitable calibration entry can be found. This is due to the fact that it is not possible to search the Frame subheaders when evaluating the rules.

flat

Return (or set) the name of the current flatfield solution.

```
$flat = $Cal->flat;
```

This method is subclassed for SCUBA-2 because we have one flatfield per subarray. Note that the user must set the subarray with the Frame class `subarray()` method before a suitable calibration entry can be found. This is due to the fact that it is not possible to search the Frame subheaders when evaluating the rules.

fastflat

Return (or set) the name of the current fast-ramp flatfield file(s).

```
$fastflat = $Cal->fastflat;
```

There is one fast-ramp flatfield file per subarray. Note that the user must set the subarray with the Frame class `subarray()` method before a suitable calibration entry can be found. This is due to the fact that it is not possible to search the Frame subheaders when evaluating the rules.

setupflat

Return the name of the matching fast-ramp flatfield file from the most recent `SETUP` observation.

```
$setupflat = $Cal->setupflat;
```

There is one fast-ramp flatfield file per subarray. Note that the user must set the subarray with the Frame class `subarray()` method before a suitable calibration entry can be found. This is due to the fact that it is not possible to search the Frame subheaders when evaluating the rules.

noise

Return the name of the matching noise observation.

```
$noise = $Cal->noise;
```

There is one noise file per subarray. Note that the user must set the subarray with the Frame class `subarray()` method before a suitable entry can be found. This is due to the fact that it is not possible to search the Frame subheaders when evaluating the rules.

nep

Return the name of the matching NEP file. This returns the name of the top-level container file - the user must then select the `.more.smurf.nep` component within that file.

```
$nep = $Cal->nep;
```

There is one noise file per subarray. Note that the user must set the subarray with the Frame class `subarray()` method before a suitable entry can be found. This is due to the fact that it is not possible to search the Frame subheaders when evaluating the rules.

zeropath

Return (or set) the name of the current zeropath file(s).

```
$zeropath = $Cal->zeropath();
```

zeropath_fwd

Return (or set) the name of the current forward zeropath file(s).

```
$zeropath = $Cal->zeropath_fwd();
```

zeropath_bck

Return (or set) the name of the current backward zeropath file(s).

```
$zeropath = $Cal->zeropath_bck();
```

General Methods

makemap_config

Return the full path to a default makemap config file. Options to return particular default configurations may be passed in as a hash. Valid keys are `config_type` and `pipeline`. The config type must be one of the supported values (see the contents of $STARLINK_DIR/share/smurf for available options). If given, the pipeline argument must be either `ql` or `summit` and causes this method to look in the directory defined by the environment variable ORAC_DATA_CAL.

```
my $config_file = $Cal->makemap_config;
my $config_file = $Cal->makemap_config( pipeline => "ql" );
```

The `config_type` is usually stored as a uhdr entry for the current Frame object.

The `pipeline` argument is ignored if the config type is given as `moon`.

pixelscale

Method to retrieve default values of the pixel scale for output images. The numbers are returned in ARCSEC. These numbers are hard-wired here and should always be retrieved with this method.

Note this is intended for use with DREAM/STARE images, not SCAN data.

resp

Return (or set) the name of the current responsivity solution.

```
$resp = $Cal->resp;
```

Note that unless the current Frame has been derived from a sub-group, the user must set the subarray with the Frame class `subarray()` method before a suitable calibration entry can be found. This is due to the fact that it is not possible to search the Frame subheaders when evaluating the rules.

respstats

Get/set the statistics associated with the most recent flatfield solution. If setting, the user must supply a subarray and hash reference with the relevant statistics. No check is made to verify the contents of the hash. The user may pass in an optional subarray argument if retrieving the stored values, otherwise the entire hash reference is returned. If so then a hash reference is returned whicih contains only the relevant info for the given subarray. A value of undef is returned if no data exist for that subarray.

```
my %allrespstats = %{ $Cal->respstats };

my $respstatsref = $Cal->respstats( $subarray );

$Cal->respstats( $subarray, \%respstats );
```

subinst

The sub-instrument associated with this calibration object. Returns either `450` or `850`.

```
$subinst = $Cal->subinst();
```

Support Methods The "mask" and "resp" methods have support implementations to obtain the index file, current name and whether the value can be updated or not. For method "cal" there will be corresponding methods "calindex", "calname" and "calnoupdate". "calcache" is an allowed synonym for "calname".

```
$current = $Cal->calcache();
$index = $Cal->calindex();
$noup = $Cal->calnoupdate();
```

SEE ALSO

ORAC::Calib::JCMTCont

COPYRIGHT

C.7 ORAC::Constants

Constants available to the ORAC system

SYNOPSIS

```
use ORAC::Constants;
use ORAC::Constants qw/ORAC__OK/;
use ORAC::Constants qw/:status/;
```

DESCRIPTION

Provide access to ORAC constants, necessary to use this module if you wish to return an ORAC__ABORT or ORAC__FATAL status using ORAC::Error.

CONSTANTS

The following constants are available from this module:

ORAC__OK

This constant contains the definition of good ORAC status.

ORAC__ERROR

This constant contains the definition of bad ORAC status.

ORAC__BADENG

An algorithm engine has returned with a status that indicates that the engine is no longer valid. This can be used to indicate that an engine has crashed and that a new one should be launched.

ORAC__ABORT

This constant contains the definition a user aborted ORAC process

ORAC__FATAL

This constant contains the definition an ORAC process which has died fatally

ORAC__PARSE_ERROR

This constant contains the definition of an error in parsing a recipe.

ORAC__TERM

This constant denotes that a recipe was terminated early, but without error.

ORAC__TERMERR

This constant denotes that a recipe was terminated early, but with a handled error.

ORAC__BADFRAME

This constant denotes that the recipe was completed with the frame marked bad.

VAL__BADD

This constant denotes the numerical value for the special "bad" value in NDF, for doubles.

TAGS

Individual sets of constants can be imported by including the module with tags. For example:

```
use ORAC::Constants qw/:status/;
```

will import all constants associated with ORAC status checking.

The available tags are:

:status

Constants associated with ORAC status checking: ORAC__OK and ORAC__ERROR.

:badvalues

Constants associated with bad values.

USAGE

The constants can be used as if they are subroutines. For example, if I want to print the value of ORAC__ERROR I can

```
use ORAC::Constants;
print ORAC__ERROR;
```

or

```
use ORAC::Constants ();
print ORAC::Constants::ORAC__ERROR;
```

SEE ALSO

constants

AUTHOR

Tim Jenness <t.jenness@jach.hawaii.edu> and Frossie Economou <frossie@jach.hawaii.edu>

REQUIREMENTS

The `constants` package must be available. This is a standard perl package.

COPYRIGHT

Copyright (C) 1998-2001 Particle Physics and Astronomy Research Council. All Rights Reserved.

C.8 ORAC::Display

Top level interface to ORAC display tools

SYNOPSIS

```
use ORAC::Display;

$Display = new ORAC::Display;
$Display->usenbs(1);
$Display->filename(filename);
$Display->display_data('frame/group object');
$Display->display_data('frame/group object',{WINDOW=>1});
```

DESCRIPTION

This module provides an OO-interface to the ORAC display manager. The display object reads device information from a file or notice board (shared memory) [NBS not implemented], determines whether the supplied frame object matches the criterion for display, if it does it instructs the relevant device object to send to the selected window (creating a new device object if necessary)

PUBLIC METHODS

Constructor

new

Create a new instance of **ORAC::Display**. No arguments are required.

```
  $Display = new ORAC::Display;
```

Accessor Methods

display_tools

Returns (or sets) a hash containing the current lookup of display tool to display tool object. For example:

```
   $Display->display_tools(%tools);
   %tools = $Display->display_tools;
```

where %tools could look like:

```
     'GAIA' => Display::GAIA=HASH(object),
     'P4'   => Display::P4=HASH(object)
```

etc. The current contents are overwritten when a new hash is supplied.

When called from an array context, returns the full hash contents. When called from a scalar context, returns the reference to the hash.

filename

Set (or retrieve) the name of the file containing the display device definition. Only used when usenbs() is false.

```
  $file = $Display->filename;
  $Display->filename("new_file");
```

idstring

Set (or retrieve) the value of the string used for comparison with the display device definition information (created by the separate device allocation GUI).

```
  $Display->idstring($id);
  $id = $Display->idstring;
```

usenbs

Determine whether NBS (shared memory) should be used to read the display device definition. Default is false.

```
$usenbs = $Display->usenbs;
$Display->usenbs(0);
```

is_master

Returns true if this object is driving a display, false if this object is monitoring a display.

monitor_handle

File handle associated with monitor file. Will be created if necessary. If this is the master, the file is opened for write (thereby removing any existing file). If this is a monitor the file is opened for read.

```
$hdl = $display->monitor_handle;
```

It is non-fatal for the monitor to fail to open a file since this usually indicates that the master pipeline is not enabling monitoring.

does_master_display

Controls whether the master display object is doing local displaying of data itself (true) or whether it is simply sending information to a monitor (false). Default is true.

title_info

Set or return information to show in window titles (where possible).

General Methods

definition

Method to read a display definition, compare it with the idstring stored in the object (this is usually a file suffix) and return back an array of hashes containing all the relevant entries from the definition. If an argument is given, the object updates its definition of current idstring (and then searches).

```
@defn = $display->definition;
@defn = $display->definition($id);
```

An empty array is returned if the suffix can not be matched.

display_data

This is the main method to be used for displaying data. The supplied object must contain a method for determining the filename and the display ID (so that it can be compared with the information stored in the device definition file). It should support the file(), nfiles() and gui_id() methods.

The optional hash can be used to supply extra entries in the display definition file (or in fact do away with the definition file completely). Note that the contents of the options hash will be used even if no display definition can be found to match the current gui_id.

```
$Display->display_data($Frm) if defined $Display;
$Display->display_data($Frm, { TOOL => 'GAIA'});
$Display->display_data($Frm, { TOOL => 'GAIA'}, $usedisp);
```

A third optional argument can be used in conjunction with the options hash to indicate whether these options should be used instead of the display definition file (false) or in addition to (true - the default). In addition this argument may take the value of -1 in which case the entries in $optref will be used to overwrite or supplement existing entries.

parse_nbs_defn

Using the current idstring, read the relevant information from a noticeboard and return it in a hash. This routine takes no arguments (idstring is read from the object) and should only be used if the usenbs() flag is true.

```
%defn = $self->parse_nbs_defn;
```

Currently not implemented.

parse_file_defn

Using the current idstring, read the relevant information from the text file (name stored in filename()) and return it in an array of hashes. There will be one hash per entry in the file that matches the given suffix. This routine takes no arguments (idstring is read from the object).

The input file is assumed to contain one line per ID of the following format:

```
ID  key=value key=value key=value..........\n
```

append_monitor

Write information to a file that can be read by monitor processes. This allows the display system to be separated from the actual tools doing the displaying or for a clone display system to enable more than one person to view pipeline output.

```
$display->append_monitor( $Frm, \%options, $usedisp );
```

Does nothing if this object is monitoring.

process_monitor_request

Given a line of text matching that written by the `append_monitor` method, trigger the display system accordingly by calling the `display_data` method.

```
$Display->process_monitor_request( $line );
```

No-op if the Display is configured as a master.

SEE ALSO

Related ORAC display devices (eg *ORAC::Display::KAPVIEW*)

COPYRIGHT

C.9 ORAC::Frame

Base class for dealing with observation frames in ORAC-DR

SYNOPSIS

```
use ORAC::Frame;

$Frm = new ORAC::Frame("filename");
$Frm->file("prefix_flat");
$num = $Frm->number;
```

DESCRIPTION

This module provides the basic methods available to all **ORAC::Frame** objects. This class should be used when dealing with individual observation files (frames).

PUBLIC METHODS

The following methods are available in this class:

Constructors The following constructors are available:

new

Create a new instance of a **ORAC::Frame** object. This method also takes optional arguments: if 1 argument is supplied it is assumed to be the name of the raw file associated with the observation. If 2 arguments are supplied they are assumed to be the raw file prefix and observation number. In any case, all arguments are passed to the configure() method which is run in addition to new() when arguments are supplied. The object identifier is returned.

```
$Frm = new ORAC::Frame;
$Frm = new ORAC::Frame("file_name");
$Frm = new ORAC::Frame("UT", "number");
```

The base class constructor should be invoked by sub-class constructors. If this method is called with the last argument as a reference to a hash it is assumed that this hash contains extra configuration information ('instance' information) supplied by sub-classes.

Note that the file format expected by this constructor is actually the required format of the data (as returned by `format()` method) and not necessarily the raw format. ORAC-DR will pre-process the data with `ORAC::Convert` prior to passing it to this constructor.

framegroup

Create new instances of objects (of this class) from multiple input files.

```
@frames = ORAC::Frame->framegroup( @files );
```

In most cases this is identical to simply passing the files directly to the constructor. In some subclasses, files from the same observation will be grouped into multiple file objects and processed independently.

Note that framegroup() accepts multiple filenames in a list, as opposed to the frame constructors that only take single files or reference to an array.

If the `framegroupkeys` method returns a list, those keys are used as FITS headers that should be used to group the input files.

subfrms

Return a list of Frame objects containing files with the given header keys. All the files in each frame will have the same values for the given keys (if they exist).

```
@subfrms = $Frm->subfrms(@keys);
```

Only the primary header is searched for subheaders (not the uhdr).

Each frame is in the same class as the given Frame.

This method is analagous to the Group `subgrps` method and will return the current Frame if there are no subheaders.

Accessor Methods The following methods are available for accessing the 'instance' data.

group

This method returns the group name associated with the observation.

```
$group_name = $Frm->group;
$Frm->group("group");
```

This can be configured initially using the findgroup() method. Alternatively, findgroup() is run automatically by the configure() method.

framegroupkeys

Returns the FITS header keys that should be used to group files into distinct Frame objects. Used by the `framegroup` method to determine grouping.

Returns UTDATE and OBSERVATION_NUMBER in base class, to allow simple disambiguation. This is required to allow the -file option to work since that relies on framegroupkeys as it reads all files at once.

If the final element of the returned list is a reference to an array those items will be treated as alternate keywords and the first keyword to match will be used.

is_frame

Whether or not the current object is an ORAC::Frame object.

```
  $is_frame = $self->is_frame;
```

Returns 1.

isgood

Flag to determine the current state of the frame. If isgood() is true the Frame is valid. If it returns false the frame object may have a problem (eg the recipe responsible for processing the frame failed to complete).

This flag is used by the **ORAC::Group** class to determine membership. A negative value indicates that the frame is good but should be hidden from the group.

The return value from isgood() will always be defined.

nsubs

Return the number of sub-frames associated with this frame.

nfiles() should be used to return the current number of sub-frames associated with the frame (nsubs usually only reports the number given in the header and may or may not be the same as the number of sub-frames currently stored)

Usually this value is set as part of the configure() method from the header (using findnsubs()) or by using findnsubs() directly.

rawfixedpart

Return (or set) the constant part of the raw filename associated with the raw data file. (ie the bit that stays fixed for every observation)

```
  $fixed = $self->rawfixedpart;
```

rawformat

Data format associated with the raw() data file. Usually one of 'NDF', 'HDS' or 'FITS'. This format should be recognisable by `ORAC::Convert`.

rawsuffix

Return (or set) the file name suffix associated with the raw data file.

```
$suffix = $self->rawsuffix;
```

recipe

This method returns the recipe name associated with the observation. The recipe name can be set explicitly but in general should be set by the findrecipe() method.

```
$recipe_name = $Frm->recipe;
$Frm->recipe("recipe");
```

This can be configured initially using the findrecipe() method. Alternatively, findrecipe() is run automatically by the configure() method.

tempraw

An array of flags, one per raw file, indicating whether the raw file is temporary, and so can be deleted, or real data (don't want to delete it).

```
$Frm->tempraw( @istemp );
@istemp = $Frm->tempraw;
```

If a single value is given, it will be applied to all raw files

```
$Frm->tempraw( 1 );
```

In scalar context returns true if all frames are temporary, false if all frames are permanent and undef if some frames are temporary whilst others are permanent.

```
$alltemp = $Frm->tempraw();
```

General Methods The following methods are provided for manipulating **ORAC::Frame** objects:

configure

This method is used to configure the object. It is invoked automatically if the new() method is invoked with an argument. The file(), raw(), readhdr(), findgroup(), findrecipe and findnsubs() methods are invoked by this command. Arguments are required. If there is one argument it is assumed that this is the raw filename. If there are two arguments the filename is constructed assuming that argument 1 is the prefix and argument 2 is the observation number.

```
$Frm->configure("fname");
$Frm->configure("UT","num");
```

Multiple raw file names can be provided in the first argument using a reference to an array.

erase

Erase the current file from disk.

```
$Frm->erase($i);
```

The optional argument specified the file number to be erased. The argument is identical to that given to the file() method. Returns ORAC__OK if successful, ORAC__ERROR otherwise.

Note that the file() method is not modified to reflect the fact the the file associated with it has been removed from disk.

This method is usually called automatically when the file() method is used to update the current filename and the nokeep() flag is set to true. In this way, temporary files can be removed without explicit use of the erase() method. (Just need to use the nokeep() method after the file() method has been used to update the current filename).

file_exists

Method to determine whether the Frame file() exists on disk or not. Returns true if the file is there, false otherwise. Effectively equivalent to using `-e` but allows for the possibility that the information stored in file() does not directly relate to the file as stored on disk (e.g. a .sdf extension). The base class is very simplistic (ie does not assume extensions).

```
$exists = $Frm->file_exists($i)
```

The optional argument refers to the file number.

file_from_bits

Determine the raw data filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
$fname = $Frm->file_from_bits($prefix, $obsnum);
```

findgroup

Returns group name from header. If we cannot find anything sensible, we return 0. The group name stored in the object is automatically updated using this value.

findnsubs

Find the number of sub-frames associated with the frame by looking in the header. Usually run by configure().

In the base class this method looks for a header keyword of 'NSUBS'.

```
$nsubs = $Frm->findnsubs;
```

The state of the object is updated automatically.

findrecipe

Method to determine the recipe name that should be used to reduce the observation. The default method is to look for an "ORAC_DR_RECIPE" entry in the user header. If one cannot be found, we assume QUICK_LOOK.

```
$recipe = $Frm->findrecipe;
```

The object is automatically updated to reflect this recipe.

files_from_hdr

Groups files in a Frame object by the value of a particular header item or items. Returns a list that can be stored in a hash with the concatenated header values as the key and a reference to an array of file names as the value. If there is no subheader all the files in the frame will be returned indexed by the concatenated values of the headers.

```
%related = $Frm->files_from_hdr( @keys )
```

Undefined values in subheaders will be treated as if they had the value "<undef>".

It is an error to call this method if the number of files in the frame does not match the number of subheaders.

flag_from_bits

Determine the name of the flag file given the variable component parts. A prefix (usually UT) and observation number should be supplied

```
$flag = $Frm->flag_from_bits($prefix, $obsnum);
```

This method should be implemented by a sub-class.

number

Method to return the number of the observation. The number is determined by looking for a number at the end of the raw data filename. For example a number can be extracted from strings of the form textNNNN.sdf or textNNNN, where NNNN is a number (leading zeroes are stripped) but not textNNNNtext (number must be followed by a decimal point or nothing at all).

```
$number = $Frm->number;
```

The return value is -1 if no number can be determined.

As an aside, an alternative approach for this method (especially in a sub-class) would be to read the number from the header.

pattern_from_bits

Determine the pattern for the raw filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
$pattern = $Frm->pattern_from_bits($prefix, $obsnum);
```

Returns a regular expression object.

template

Method to change the current filename of the frame (file()) so that it matches a template. e.g.:

```
  $Frm->template("something_number_flat");
```

Would change the first file to match "something_number_flat". Essentially this simply means that the number in the template is changed to the number of the current frame object.

```
  $Frm->template("something_number_dark", 2);
```

would change the second filename to match "something_number_dark". The base method assumes that the filename matches the form: prefix_number_suffix. This must be modified by the derived classes since in general the filenaming convention is telescope and instrument specific.

The Nth filename is modified (ie file(N)). There are no return arguments.

SEE ALSO

ORAC::Group

COPYRIGHT

C.10 ORAC::Frame::NDF

Class for dealing with frames based on NDF files

SYNOPSIS

```
  use ORAC::Frame::NDF

  $Frm = new ORAC::Frame::NDF;
```

DESCRIPTION

This class provides implementations of the methods that require knowledge of the NDF file format rather than generic methods or methods that require knowledge of a specific instrument. In general, the specific instrument sub-classes will inherit from the file type (which inherits from ORAC::Frame) rather than directly from ORAC::Frame. For JCMT and UKIRT the group files are based on NDFs and inherit from this class.

The format specific sub-classes do not contain constructors; they should be defined in either the base class or the instrument specific sub-class.

PUBLIC METHODS

The following methods are modified from the base class versions.

General Methods

erase

Erase the current file from disk.

```
$Frm->erase($i);
```

The optional argument specifies the file number to be erased. The argument is identical to that given to the file() method. Returns ORAC__OK if successful, ORAC__ERROR otherwise.

Note that the file() method is not modified to reflect the fact the the file associated with it has been removed from disk.

This method is usually called automatically when the file() method is used to update the current filename and the nokeep() flag is set to true. In this way, temporary files can be removed without explicit use of the erase() method. (Just need to use the nokeep() method after the file() method has been used to update the current filename).

Can support paths to HDS objects. If the last object is removed from an HDS container file, the entire container file is removed.

file_exists

Checks for the existence of the frame file(). Assumes a `.sdf` extension.

```
$exists = $Frm->exists($i)
```

The optional argument specifies the file number to be used. All extension are removed from the file name before adding the `.sdf` so that HDS containers can be supported (and files that already have the extension) -- but note that this version of the method does not look inside HDS containers looking for NDFs.

inout

Method to return the current input filename and the new output filename given a suffix. Copes with non-existence of HDS container and handles NDF subframes on the assumption that any NDF with a "." in it must be referring to an HDS component.

The suffix is appended to the root filename derived from the characters before the first ".". Note that this uses the ORAC-DR standard, replacing the first non-numeric suffix.

The following logic is applied when propogating HDS containers:

```
 - If a '.' is present

   NFILES > 1
       The new suffix is attached before the dot.
       An HDS container is created (based on the root) to
       receive the expected NDF. If present, a .HEADER component
       is copied to the output file, else, the current FITS header
       is written to a .HEADER component. Note that even if multiple
       HDS component paths are included in the input file, only
       the last section of the path is copied to the output file.
       ie file.A.B.I1 will result in outfile.I1

   NFILES = 1
       We remove the dot and append the suffix as normal
       (by removing the old suffix first).
       This ensures that when NFILES=1 we will no longer
       be using HDS containers

 - If no '.' is present

       This is the standard behaviour. Simply remove after
       last underscore and replace with new suffix.
```

If you want to retain the HDS container syntax, this routine has to be fooled into thinking that nfiles is greater than 1 (eg by adding a dummy file name to the frame).

Returns $out in a scalar context:

```
$out = $Frm->inout($suffix);
```

Returns $in and $out in an array context:

```
($in, $out) = $Frm->inout($suffix);

($in,$out) = $Frm->inout($suffix,2);
```

The second (optional) argument is used to specify which of the input filenames should be used to generate an output name. This number is forwarded to the file() method and defaults to 1 (ie the first frame).

If a value of 0 is provided, the output name is derived assuming the NFILES=1 rule described above. This allows the output file name to be derived correctly in the many-to-one scenario.

SEE ALSO

ORAC::Frame, *ORAC::BaseNDF*

COPYRIGHT

C.11 ORAC::Frame::UKIRT

UKIRT class for dealing with observation files in ORAC-DR

SYNOPSIS

```
use ORAC::Frame::UKIRT;

$Frm = new ORAC::Frame::UKIRT("filename");
$Frm->file("file")
$Frm->readhdr;
$Frm->configure;
$value = $Frm->hdr("KEYWORD");
```

DESCRIPTION

This module provides methods for handling Frame objects that are specific to UKIRT. It provides a class derived from **ORAC::Frame::NDF**. All the methods available to **ORAC::Frame** objects are available to **ORAC::Frame::UKIRT** objects.

PUBLIC METHODS

The following methods are available in this class in addition to those available from **ORAC::Frame**.

General Methods

file_from_bits

Determine the raw data filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
  $fname = $Frm->file_from_bits($prefix, $obsnum);
```

The $obsnum is zero padded to 5 digits.

pattern_from_bits() is currently an alias for file_from_bits(), and both can be used interchangably in the UKIRT subclass.

flag_from_bits

Determine the name of the flag file given the variable component parts. A prefix (usually UT) and observation number should be supplied

```
  $flag = $Frm->flag_from_bits($prefix, $obsnum);
```

This generic UKIRT version returns back the observation filename (from file_from_bits) , adds a leading "." and replaces the .sdf with .ok

inout

Method to return the current input filename and the new output filename given a suffix. Copes with non-existence of HDS container and handles NDF subframes

The following logic is applied:

```
 - If a '.' is present

   NFILES > 1
       The new suffix is attached before the dot.
       An HDS container is created (based on the root) to
       receive the expected NDF.

   NFILES = 1
       We remove the dot and append the suffix as normal
       (by removing the old suffix first).
       This ensures that when NFILES=1 we will no longer
       be using HDS containers

 - If no '.' is present

       This is the standard behaviour. Simply remove after
       last underscore and replace with new suffix.
```

If you want to retain the HDS container syntax, this routine has to be fooled into thinking that nfiles is greater than 1 (eg by adding a dummy file name to the frame).

Returns $out in a scalar context:

```
        $out = $Frm->inout($suffix);
```

Returns $in and $out in an array context:

```
        ($in, $out) = $Frm->inout($suffix);

        ($in,$out) = $Frm->inout($suffix,2);
```

SEE ALSO

ORAC::Group, ORAC::Frame::NDF, ORAC::Frame

COPYRIGHT

C.12 ORAC::Frame::WFCAM

WFCAM class for dealing with observation files in ORAC-DR with Starlink software.

SYNOPSIS

```
  use ORAC::Frame::WFCAM;

  $Frm = new ORAC::Frame::FWCAM("filename");
  $Frm->file("file");
  $Frm->readhdr;
  $Frm->configure;
  $value = $Frm->hdr("KEYWORD");
```

DESCRIPTION

This module provides methods for handling Frame objects that are specific to WFCAM, allowing them to be reduced using Starlink software. It provides a class derived from **ORAC::Frame::WFCAM**. All the methods available to **ORAC::Frame::WFCAM** objects are available to **ORAC::Frame::WFCAM** objects. Some additional methods are supplied.

PUBLIC METHODS

The following methods are available in this class in addition to those available from **ORAC::Frame::WFCAM**.

Constructor

new

Create a new instance of a **ORAC::Frame::WFCAM** object. This method also takes optional arguments: if 1 argument is supplied it is assumed to be the name of the raw file associated with the observation. If 2 arguments are supplied they are assumed to be the raw file prefix and observation number. In any case, all arguments are passed to the configure() method which is run in addition to new() when arguments are supplied. The object identifier is returned.

```
$Frm = new ORAC::Frame::WFCAM;
$Frm = new ORAC::Frame::WFCAM("file_name");
$Frm = new ORAC::Frame::WFCAM("UT","number");
```

The constructor hard-wires the '.sdf' rawsuffix and the prefix although these can be overriden with the rawsuffix() and rawfixedpart() methods. The prefix depends on the value of the ORAC_INSTRUMENT environment variable; if this is set to WFCAM1, WFCAM2, WFCAM3, or WFCAM4, then the prefix is set to 'w', 'x', 'y', or 'z', respectively. Otherwise the prefix defaults to 'w'.

General Methods

file_from_bits

Determine the raw data filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
$fname = $Frm->file_from_bits($prefix, $obsnum);
```

mergehdr

Method to propagate the FITS header from an HDS container to an NDF Run after updating $Frm.

```
$Frm->files($out);
$Frm->mergehdr;
```

number

Method to return the number of the observation. The number is determined by looking for a number at the end of the raw data filename. For example a number can be extracted from strings of the form textNNNN.sdf or textNNNN, where NNNN is a number (leading zeroes are stripped) but not textNNNNtext (number must be followed by a decimal point or nothing at all).

```
$number = $Frm->number;
```

The return value is -1 if no number can be determined.

As an aside, an alternative approach for this method (especially in a sub-class) would be to read the number from the header.

PRIVATE METHODS

_split_name

Internal routine to split a 'file' name into an actual filename (the HDS container) and the NDF name (the thing inside the container).

Splits on '.'

Argument: string to split (eg test.i1) Returns: root name, ndf name (eg 'test' and 'i1')

NDF name is undef if there are no 'sub-frames'.

This routine is so simple that it may not be worth the effort.

OLD CASU STUFF

phukeys

Returns the list of primary header unit keywords @phukeys = $Frm->phukeys;

ehukeys

Returns the list of extension header unit keywords @ehukeys = $Frm->ehukeys;

SEE ALSO

ORAC::Group

COPYRIGHT

C.13 ORAC::Frame::JCMT

JCMT class for dealing with observation files in ORAC-DR.

SYNOPSIS

```
use ORAC::Frame::JCMT;

$Frm = new ORAC::Frame::JCMT( "filename" );
```

DESCRIPTION

This module provides methods for handling Frame objects that are specific to JCMT instruments. It provides a class derived from **ORAC::Frame::NDF**. All the methods available to **ORAC::Frame** objects are also available to **ORAC::Frame::JCMT** objects.

PUBLIC METHODS

The following methods are available in this class in addition to those available from **ORAC::Frame**.

Accessors

allow_header_sync

Whether or not to allow automatic header synchronization when the Frame is updated via either the `file` or `files` method.

```
$Frm->allow_header_sync( 1 );
```

For modern JCMT instruments, defaults to true (1).

General Methods

jcmtstate

Return a value from either the first or last entry in the JCMT STATE structure.

```
my $value = $Frm->jcmtstate( $keyword, 'end' );
```

If the supplied keyword does not exist in the JCMT STATE structure, this method returns undef. An optional second argument may be given, and must be either 'start' or 'end'. If this second argument is not given, then the first entry in the JCMT STATE structure will be used to obtain the requested value.

Both arguments are case-insensitive.

find_base_position

Determine the base position of a data file. If the file name is not provided it will be read from the object.

```
%base = $Frm->find_base_position( $file );
```

Returns hash with keys

```
TCS_TR_SYS   Tracking system for base
TCS_TR_BC1   Longitude of base position (radians)
TCS_TR_BC2   Latitude of base position (radians)
```

The latter will be absent if this is an observation of a moving source. In addition, returns sexagesimal strings of the base position as

```
TCS_TR_BC1_STR
TCS_TR_BC2_STR
```

findgroup

Returns the group name from the header or a string formed automatically on observation metadata.

```
$Frm->findgroup();
```

An optional argument can be provided which will be appended to the group name. This can be used by subclasses to provide additional information required to disambiguate groups. This string is only used if the group identifier is not present in the DRGROUP header.

```
$Frm->findgroup( $string );
```

The group name stored in the object is automatically update using this value.

findnsubs

Find the number of sub-frames associated by the frame by looking at the list of raw files associated with object. Usually run by configure().

```
$nsubs = $Frm->findnsubs;
```

The state of the object is updated automatically.

SEE ALSO

ORAC::Group, *ORAC::Frame::NDF*, *ORAC::Frame*

COPYRIGHT

Copyright (C) 2009 Science and Technology Facilities Council. All Rights Reserved.

C.14 ORAC::Frame::ACSIS

Class for dealing with ACSIS observation frames.

SYNOPSIS

use ORAC::Frame::ACSIS;

$Frm = new ORAC::Frame::ACSIS(\@filenames); $Frm->file("file"); $Frm->readhdr; $Frm->configure; $value = $Frm->hdr("KEYWORD");

DESCRIPTION

This module provides methods for handling Frame objects that are specific to ACSIS. It provides a class derived from **ORAC::Frame::NDF**. All the methods available to **ORAC::Frame** objects are available to **ORAC::Frame::IRIS2** objects.

PUBLIC METHODS

The following methods are available in this class in addition to those available from **ORAC::Frame**.

Constructor

new

Create a new instance of an **ORAC::Frame::ACSIS** object. This method also takes optional arguments:

- If one argument is supplied it is assumed to be a reference to an array containing a list of raw files associated with the observation.
- If two arguments are supplied they are assumed to be the UT date and observation number.

In any case, all arguments are passed to the configure() method which is run in addition to new() when arguments are supplied.

The object identifier is returned.

```
$Frm = new ORAC::Frame::ACSIS;
$Frm = new ORAC::Frame::ACSIS( \@files );
$Frm = new ORAC::Frame::ACSIS( '20040919', '10' );
```

The constructor hard-wires the '.sdf' rawsuffix and the 'a' prefix, although these can be overridden with the rawsuffix() and rawfixedpart() methods.

configure

This method is used to configure the object. It is invoked automatically if the new() method is invoked with an argument. The file(), raw(), readhdr(), findgroup(), findrecipe() and findnsubs() methods are invoked by this command. Arguments are required. If there is one argument it is assumed that this is a reference to an array containing a list of raw filenames. The ACSIS version of configure() cannot take two parameters, as there is no way to know the location of the file that would make up the Frame object from only the UT date and run number.

```
$Frm->configure(\@files);
```

framegroupkeys

Returns the keys that should be used for determining whether files from a single observation should be treated independently.

For ACSIS a single frame object is returned for single sub-system observations and multiple frame objects returned in multi-subsystem mode. One caveat is that if the multi-subsystem mode looks like a hybrid mode (bandwidth mode and IF frequency identical) then a single frame object is returned.

```
@keys = $Frm->framegroupkeys;
```

This implementation includes an additional reference to an array containing alternate keys. ACSIS data uses IFFREQ to determine whethere subbands should be merged whereas DAS data uses SPECID.

General Methods

file_from_bits

There is no file_from_bits() for ACSIS. Use pattern_from_bits() instead.

file_from_bits_extra

Extra information that can be supplied to the Group file_from_bits methods when constructing the Group filename.

```
$extra = $Frm->file_from_bits_extra();
```

flag_from_bits

Determine the name of the flag file given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
 $flag = $Frm->flag_from_bits($prefix, $obsnum);
```

For ACSIS the flag file is of the form .aYYYYMMDD_NNNNN.ok, where YYYYMMDD is the UT date and NNNNN is the observation number zero-padded to five digits. The flag file is stored in $ORAC_DATA_IN.

findgroup

Returns the group name from the header.

The group name stored in the object is automatically updated using this value.

inout

Similar to base class except the frame number is appended to the output suffix.

jsa_pub_asn_id

Determine the association ID to be used for the JCMT Science Archive to collect the "public" products. This is written in plain text as it will be short enough to not require an md5sum to be taken as is the case for `asn_id()`.

This contains the components necessary to distinguish the desired "public" co-adds as determined by the instrument scientist. It does not contain the tile number. Instead it identifies the whole association -- i.e. all the public data for a particular set of configurations which we can co-add. The wrapper script (*jsawrapdr*) will call the `JSA::Headers::CADC::correct_asn_id()` subroutine to add the tile number to this at a later stage.

Returns "undef" on failure (e.g. for an unsupported instrument or for configurations which the instrument scientist has decided to reject).

pattern_from_bits

Determine the pattern for the raw filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
  $pattern = $Frm->pattern_from_bits( $prefix, $obsnum );
```

Returns a regular expression object.

number

Method to return the number of the observation. The number is determined by looking for a number after the UT date in the filename. This method is subclassed for ACSIS.

The return value is -1 if no number can be determined.

subsystem_id

Subsystem identifier. For ACSIS this is the rest frequency, bandwidth mode and first subsystem number.

<SPECIALIST METHODS>

Methods specifically for ACSIS.

subsysnrs

List of subsysnumbers in use for this frame. If there is more than one subsystem number this indicates a hybrid mode.

```
  @numbers = $Frm->subsysnrs;
```

In scalar context returns the total number of subsystems.

```
  $number_of_subsystems = $Frm->subsysnrs;
```

rest_frequency

Attempt to determine the rest frequency. Returns undef on failure.

Intended to maintain the historical behavior of `findgroup` in order that the nightly association IDs do not change:

```
    my $rf = $self->rest_frequency(0);
```

The first parameter can be set to a true value to allow the rest frequency to be determined by averaging FRQSIGLO and FRQSIGHI. This is just for the historical behavior mentioned above, and should not be used if a real rest frequency is required.

SEE ALSO

ORAC::Frame::NDF

COPYRIGHT

C.15 ORAC::Frame::SCUBA

SCUBA class for dealing with observation files in ORACDR

SYNOPSIS

```
use ORAC::Frame::SCUBA;


$Frm = new ORAC::Frame::SCUBA("filename");
$Frm->file("file")
$Frm->readhdr;
$Frm->configure;
$value = $Frm->hdr("KEYWORD");
```

DESCRIPTION

This module provides methods for handling Frame objects that are specific to SCUBA. It provides a class derived from **ORAC::Frame**. All the methods available to **ORAC::Frame** objects are available to **ORAC::Frame::SCUBA** objects. Some additional methods are supplied.

PUBLIC METHODS

The following are modifications to standard ORAC::Frame methods.

Constructors

new

Create a new instance of a **ORAC::Frame::SCUBA** object. This method also takes optional arguments: if 1 argument is supplied it is assumed to be the name of the raw file associated with the observation. If 2 arguments are supplied they are assumed to be the raw file prefix and observation number. In any case, all arguments are passed to the configure() method which is run in addition to new() when arguments are supplied. The object identifier is returned.

```
$Frm = new ORAC::Frame::SCUBA;
$Frm = new ORAC::Frame::SCUBA("file_name");
$Frm = new ORAC::Frame::SCUBA("UT","number");
```

This method runs the base class constructor and then modifies the rawsuffix and rawfixedpart to be '.sdf' and '_dem_' respectively.

Subclassed methods The following methods are provided for manipulating **ORAC::Frame::SCUBA** objects. These methods override those provided by **ORAC::Frame**.

calc_orac_headers

This method calculates header values that are required by the pipeline by using values stored in the header.

```
%new = $Frm->calc_orac_headers();
```

This method calculates WVM statistics in addition to calling the standard base class calculations.

This method updates the frame header. Returns a hash containing the new keywords.

configure

This method is used to configure the object. It is invoked automatically if the new() method is invoked with an argument. The file(), raw(), readhdr(), findgroup(), findrecipe(), findsubs() findfilters() and findwavelengths() methods are invoked by this command. Arguments are required. If there is one argument it is assumed that this is the raw filename. If there are two arguments the filename is constructed assuming that arg 1 is the prefix and arg2 is the observation number.

```
$Frm->configure("fname");
$Frm->configure("UT","num");
```

The sub-instrument configuration is also stored.

file_from_bits

Determine the raw data filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
  $fname = $Frm->file_from_bits($prefix, $obsnum);
```

pattern_from_bits() is currently an alias for file_from_bits(), and both can be used interchangably for SCUBA.

flag_from_bits

Determine the flag filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
  $fname = $Frm->file_from_bits($prefix, $obsnum);
```

The format is ".20021001_dem_0001"

findgroup

Return the group associated with the Frame. This group is constructed from header information. The group name is automatically updated in the object via the group() method.

The group membership can be set using the DRGROUP keyword in the header. If this keyword exists and is not equal to 'UNKNOWN' the contents will be returned.

Alternatively, if DRGROUP is not specified the group name is constructed from the MODE, OBJECT and FILTER keywords. This may cause problems in the following cases:

```
 - The chop throw changes and the data should not be coadded
 [in general this is true except for LO chopping scan maps
 where all 6 chops should be included in the group]

 - The source name is the same, the mode is the same and the
 filter is the same but the source coordinates are different by
 a degree or more. In some cases [a large scan map] these should
 be in the same group. In other cases they probably should not
 be. Should I worry about it? One example was where the observer
 used RB coordinates by mistake for a first map and then changed
 to RJ -- the coordinates and source name were identical but the
 position on the sky was miles off. Maybe this should be dealt with
 by using the Frame ON/OFF facility [so it would be part of the group
 but the observer would turn the observation off]

 - Different source names are being used for offsets around
 a common centre [eg the Galactic Centre scan maps]. In this case
 we do want to coadd but this means we should be using position
 rather than source name. Also, how do we define when two fields
 are too far apart to be coadded

 - Photometry data should never be in the same group as a source
 that has a different pointing centre. Note this really should take
 MAP_X and MAP_Y into account since data should be of the same group
 if either the ra/dec is given or if the mapx/y is given relative
 to a fixed ra/dec.
```

Bottom line is the following (I think).

In all cases the actual position in RJ coordinates should be calculated (taking into account RB->RJ and GA->RJ and map_x map_y, local_coords) using Astro::PAL. Filter should also be matched as now. Planets will be special cases - matching on name rather than position.

PHOTOM observations

```
Should match positions exactly (within 1 arcsec). Should also match
chop throws [since the gain is different]. The observer is responsible
for a final coadd. Source name then becomes irrelevant.
```

JIGGLE MAP

```
Should match positions to within 10 arcmin (say). Should match chop
throw.
```

SCAN MAP

```
Should match positions to 1 or 2 degrees?
Should ignore chop throws (the primitive deals with that).
```

The group name will then use the position with a number of significant figures changing depending on the position tolerance.

findnsubs

Forces the object to determine the number of sub-instruments associated with the data by looking in the header (hdr()). The result is stored in the object using nsubs().

Unlike findgroup() this method will always search the header for the current state.

findrecipe

Return the recipe associated with the frame. The state of the object is automatically updated via the recipe() method.

The recipe is determined by looking in the FITS header of the frame. If the 'DRRECIPE' is present and not set to 'UNKNOWN' then that is assumed to specify the recipe directly. Otherwise, header information is used to try to guess at the reduction recipe. The default recipes are keyed by observing mode:

```
SKYDIP => 'SCUBA_SKYDIP'
NOISE  => 'SCUBA_NOISE'
POINTING => 'SCUBA_POINTING'
PHOTOM => 'SCUBA_STD_PHOTOM'
JIGMAP => 'SCUBA_JIGMAP'
JIGMAP (phot) => 'SCUBA_JIGPHOTMAP'
EM2_SCAN => 'SCUBA_EM2SCAN'
EKH_SCAN => 'SCUBA_EKHSCAN'
JIGPOLMAP => 'SCUBA_JIGPOLMAP'
SCANPOLMAP => 'SCUBA_SCANPOLMAP'
ALIGN  => 'SCUBA_ALIGN'
FOCUS  => 'SCUBA_FOCUS'
```

So called "wide" photometry is treated as a map (although this depends on the name of the jiggle pattern which may change).

In future we may want to have a separate text file containing the mapping between observing mode and recipe so that we dont have to hard wire the relationship.

template

This method is identical to the base class template method except that only files matching the specified sub-instrument are affected.

```
$Frm->template($template, $sub);
```

If no sub-instrument is specified then the first file name is modified

Note that this is different from the base class which accepts a file number as the second argument. This may need some rationalisation.

NEW METHODS FOR SCUBA

This section describes methods that are available in addition to the standard methods found in **ORAC::Frame**.

Accessor Methods The following extra accessor methods are provided:

filters

Return or set the filter names associated with each sub-instrument in the frame.

subs

Return or set the names of the sub-instruments associated with the frame.

wavelengths

Return or set the wavelengths associated with each sub-instrument in the frame.

New methods The following additional methods are provided:

file2sub

Given a file index, (see file()) returns the associated sub-instrument.

```
$sub = $Frm->file2sub(2)
```

Returns the first sub name if index is too large. This assumes that the file names associated wth the object are linked to sub-instruments (as returned by the subs method). It is up to the primitive writer to make sure that subs() tracks changes to files().

findfilters

Forces the object to determine the names of all sub-instruments associated with the data by looking in the hdr().

The result is stored in the object using filters(). The sub-inst filter name is made to match the filter name such that a filter of ′450w:850w′ has filter names of ′450W′ and ′850W′ despite the entries in the header being simply ′450′ and ′850′. Photometry filter names are not modified.

Unlike findgroup() this method will always search the header for the current state.

findsubs

Forces the object to determine the names of all sub-instruments associated with the data by looking in the header (hdr()). The result is stored in the object using subs().

Unlike findgroup() this method will always search the header for the current state.

findwavelengths

Forces the object to determine the names of all sub-instruments associated with the data by looking in the header (hdr()). The result is stored in the object using wavelengths().

Unlike findgroup() this method will always search the header for the current state.

sub2file

Given a sub instrument name returns the associated file index. This is the reverse of sub2file. The resulting value can be used directly in file() to retrieve the file name.

```
  $file = $Frm->file($Frm->sub2file('LONG'));
```

A case insensitive comparison is performed.

Returns 1 if nothing matched (ie just returns the first file in file(). This is probably a bug.

Assumes that changes in subs() are reflected in files().

SEE ALSO

ORAC::Frame, *ORAC::Frame::NDF*

COPYRIGHT

C.16 ORAC::Frame::SCUBA2

SCUBA-2 class for dealing with observation files in ORACDR

SYNOPSIS

```
use ORAC::Frame::SCUBA2;

$Frm = new ORAC::Frame::SCUBA2("filename");
$Frm = new ORAC::Frame::SCUBA2(@files);
$Frm->file("file")
$Frm->readhdr;
$Frm->configure;
$value = $Frm->hdr("KEYWORD");
```

DESCRIPTION

This module provides methods for handling Frame objects that are specific to SCUBA-2. It provides a class derived from **ORAC::Frame**. All the methods available to **ORAC::Frame** objects are available to **ORAC::Frame::SCUBA2** objects. Some additional methods are supplied.

PUBLIC METHODS

The following are modifications to standard ORAC::Frame methods.

Constructors

new

Create a new instance of a **ORAC::Frame::SCUBA2** object. This method also takes optional arguments: if 1 argument is supplied it is assumed to be the name of the raw file associated with the observation but if a reference to an array is supplied, each file listed in the array is used. If 2 arguments are supplied they are assumed to be the raw file prefix and observation number. In any case, all arguments are passed to the configure() method which is run in addition to new() when arguments are supplied. The object identifier is returned.

```
$Frm = new ORAC::Frame::SCUBA2;
$Frm = new ORAC::Frame::SCUBA2("file_name");
$Frm = new ORAC::Frame::SCUBA2(\@files);
$Frm = new ORAC::Frame::SCUBA2("UT","number");
```

This method runs the base class constructor and then modifies the rawsuffix and rawfixed-part to be '.sdf' and 's4' or 's8' (depending on instrument designation) respectively.

Subclassed methods The following methods are provided for manipulating **ORAC::Frame::SCUBA2** objects. These methods override those provided by **ORAC::Frame**.

configure

Configure the frame object. Usually called from the constructor.

Can be called either with a single filename or a reference to an array of filenames

```
  $Frm->configure( \@files );
  $Frm->configure( $file );
```

framegroupkeys

For SCUBA-2 a single frame object is returned in most cases. For focus observations each focus position is returned as a separate Frame object. This simplifies the recipes and allows the QL and standard recipes to work in the same way.

Observation ID is also kept separate in case the pipeline gets so far behind that the system detects the end of one observation and the start of the next.

```
 @keys = $Frm->framegroupkeys;
```

file_from_bits

Determine the raw data filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
  $fname = $Frm->file_from_bits($prefix, $obsnum);
```

Not implemented for SCUBA-2 because of the multiple files that can be associated with a particular UT date and observation number: the multiple sub-arrays (a to d) and the multiple subscans.

file_from_bits_extra

Method to return `extra` information to be used in the file name. For SCUBA-2 this is a string representing the wavelength. See also the `subsystem_id` method which currently returns the same thing but with units included.

```
  my $extra = $Frm->file_from_bits_extra;
```

inout

Acts like the base class `inout` method unless the `uhdr` entry `ALLOW_NUMBERED_SUFFICES` is set, in which case, if the file already has a suffix including a 3-digit number, that number is preserved in the suffix of the new file.

This is not quite the same as the corresponding method in the ACSIS frame class because we don’t assume that the number in the suffix is the same as the file number within the frame -- there could be multiple files with the same suffix number, e.g. from different subarrays.

pattern_from_bits

Determines the pattern for the flag file. This differs from other instruments in that SCUBA-2 writes the flag files to ORAC_DATA_IN but the data are written to completely distinct trees (not sub directories of ORAC_DATA_IN).

```
  $pattern = $Frm->pattern_from_bits( $prefix, $obsnum );
```

Returns a regular expression object.

number

Method to return the number of the observation. The number is determined by looking for a number after the UT date in the filename. This method is subclassed for SCUBA-2 to deal with SCUBA-2-specific filenames.

The return value is -1 if no number can be determined.

subsystem_id

Method to return the subsystem identifier. For SCUBA-2 this is a string representing the wavelength.

```
  my $ssid = $self->subsystem_id();
```

flag_from_bits

Determine the flag filename given the variable component parts. A prefix (usually UT) and observation number should be supplied.

```
  @fnames = $Frm->file_from_bits($prefix, $obsnum);
```

Returns multiple file names (one for each array) and throws an exception if called in a scalar context. The filename returned will include the path relative to ORAC_DATA_IN, where ORAC_DATA_IN is the directory containing the flag files.

The format is "swxYYYYMMDD_NNNNN.ok", where "w" is the wavelength signifier ('8' for 850 or '4' for 450) and "x" a letter from 'a' to 'd'.

findgroup

Return the group associated with the Frame. This group is constructed from header information. The group name is automatically updated in the object via the group() method.

get_files_by_subarray

Return a hash with subarray names as keys and values containing a list (array reference) of the associated file names.

```
  my %subarrayfiles = $Frm->get_files_by_subarray;
```

instap_subarray

Return the name of the subarray which corresponds to the instrument aperture at the current wavelength.

```
    my $data_instap = $Frm->instap_subarray;
```

jsa_pub_asn_id

Determine the association ID to be used for the JCMT Science Archive to collect the "public" products. This is written in plain text as it will be short enough to not require an md5sum to be taken as is the case for `asn_id()`.

Currently the same as `subsystem_id` but implemented separately so that changes to one do not alter the other.

meta_file

Search for and return the full path to the meta file associated with the given observation. The meta file is written by the SCUBA-2 data acquisition system and contains the name of the flag files (.ok) in use. The naming convention is "sw_YYYYMMDD_NNNNN.meta", where "w" is the wavelength signifier ('8' for 850 or '4' for 450).

Takes two mandatory arguments which are the file prefix (usually the UT date) and the observation number. Returns undef if no such file exists.

```
    my $meta_file = $self->meta_file( $prefix, $obsnum );
```

numsubarrays

Return the number of subarrays in use. Works by checking for unique subheaders and determining which of the abcd sub-arrays are producing data. Only works once data are read in so ORAC-DR must have some other way of knowing that there are n subarrays.

rewrite_outfile_subarray

This method modifies the supplied filename to remove specific subarray designation and replace it with a generic filter designation. 4 digit subscan information is also removed but not if this is a Focus observation or if the data are a fast-ramp flatfield (as there may be multiple fastflats in a single observation).

Should be used when subarrays are merged into a single file.

If the s8a/s4a designation is missing the filename will be returned unchanged.

```
    $outfile = $Frm->rewrite_outfile_subarray( $old_outfile );
```

The output file, or output HDS extension in a container, will be removed by this routine. If no HDS container exists and it looks like one will be required, it is created.

strip_subscan

Strip subscan number from supplied filename. This can be used when going from a group of files from a single subarray to a file associated with that subarray but no subscan.

```
   $new = $Frm->strip_subscan( $old );
```

subarray

Return the name of the subarray given either a filename or the index into a Frame object. The subarray is also stored in the current Frame uhdr. Takes a mandatory sole argument which is the file index or name. If the Frame header has subheaders then the subarray is obtained from the FITS header of the file rather than assuming the subheader order is synchronized with that of the files. Returns undef if the FITS header could not be retrieved.

```
my $subarray = $Frm->subarray( $i );
my $subarray = $Frm->subarray( $Frm->file($i) );
$Frm->subarray( $file );
```

Works on the current frame only.

subarrays

Return a list of the subarrays associated with the current Frame object. Searches the subheaders for presence of SUBARRAY keyword which will be present if data from multiple subarrays are stored in the current Frame. If not found then just use the SUBARRAY entry in the hdr.

```
@subarrays = $Frm->subarrays;
```

Returns an list. In scalar context returns the number of subarrays.

makemap_args

Return a list of supported arguments for makemap which may be specified as recipe parameters. Returns array or array reference depending on calling context.

```
my @makemap_args = $Frm->makemap_args;
```

The supported arguments are (currently): alignsys, config, crota, maxmem, method, pixsize, spread, system.

get_fastramp_flats

Retrieve files in the current Frame which have a sequence type of `FASTFLAT`.

```
my @fastramps = $Frm->get_fastramp_flats;
```

Returns an array, which will be empty if no fast-ramp flatfield files could be found.

duration_science

The difference between the end of the last science sequence and the start of the first science sequence. This is not quite the same as the duration of the observation because it will not include initial/trailing darks and ramps.

```
$dur = $Frm->duration_science();
```

Returns the duration in seconds.

To calculate the duration of the observation as a whole use ORAC_UTSTART and ORAC_UTEND uhdrs.

filter_darks

Standard image-based DREAM and STARE processing has no need for dark frames and so these should be filtered out as early as possible to prevent weird errors. The translated header for the observation mode is used. From this point on, no dark frames will be returned unless the user accesses the raw data.

```
$Frm->filter_darks;
```

SEE ALSO

ORAC::Frame, *ORAC::Frame::NDF*

COPYRIGHT

C.17 ORAC::General

Simple perl subroutines that may be useful for primitives

SYNOPSIS

```
use ORAC::General;

$max = max(@values);
$min = min(@values);
$result = log10($value);
$result = nint($value);
$yyyymmdd = utdate;
%hash = parse_keyvalues($string);
@obs = parse_obslist($string);
$result = cosdeg( 45.0 );
$result = sindeg( 45.0) ;
@dms = dectodms( $dec );

print "Is a number" if is_numeric( $number );
```

DESCRIPTION

This module provides simple perl functions that are not available from standard perl. These are available to all ORAC primitive writers, but they are general in nature and have no connection to orac. Some of these are used in the ORAC infastructure, so ORACDR does require this library in order to run.

SUBROUTINES

cosdeg

Return the cosine of the angle. The angle must be in degrees.

sindeg

Return the sine of the angle. The angle must be in degrees.

dectodms

Convert decimal angle (degrees or hours) to degrees, minutes and seconds. (or hours).

```
($deg, $min, $sec) = dectodms( $decimal );
```

hmstodec

Convert hours:minutes:seconds to decimal hours.

```
my $hms = "23:58:01.23";
my $dec = hmstodec($hms);
```

is_numeric

Determine whether the supplied argument is a number or not. Returns true if it is and false otherwise.

min

Find the minimum value of an array. Can also be used to find the minimum of a list of scalars since arguments are passed into the subroutine in an array context.

```
$min = min(@values);
$min = min($a, $b, $c);
```

max

Find the maximum value of an array. Can also be used to find the maximum of a list of scalars since arguments are passed into the subroutine in an array context.

```
$max = max(@values);
$max = max($a, $b, $c);
```

log10

Returns the logarithm to base ten of a scalar.

```
$value = log10($number);
```

Currently uses the implementation of log10 found in the POSIX module

nint

Return the nearest integer to a supplied floating point value. 0.5 is rounded up.

utdate

Return the UT date (strictly, GMT) date in the format yyyymmdd

parse_keyvalues

Takes a string of comma-separated key-value pairs and return a hash.

```
%hash = parse_keyvalues("a=1,b=2,C=3");
```

The keys are down-cased.

Values can be quoted or bracketed. If values include commas themselves they will be returned as array references, e.g.

```
"a=1,b=2,3,c='4,5',d=[1,2,3,4],e=[5]"
```

will return the following.

```
 a => 1,
 b => [2,3],
 c => [4,5],
 d => [1,2,3,4],
 e => 5,
```

Note that delimiters are removed from the values and that if only a single element is quoted it will be returned as a scalar string rather than an array.

parse_obslist

Converts a comma separated list of observation numbers (as supplied on the command line for the -list option) and converts it to an array of observation numbers. Colons are treated as range arguments.

For example,

```
"5,9:11"
```

is converted to

```
(5,9,10,11)
```

convert_args_to_string

Convert a hash as returned by ORAC::Recipe::PrimitiveParser->_parse_prim_arguments into a string that can be output for logging.

```
my $str = convert_args_to_string( $args );
```

Frame objects passed in will be stringified to "Frame::<class>". Group objects passed in will be stringified to "Group::<class>". Undefined variables will be stringified to "undef".

read_file_list

Given either a filename or an ORAC::TempFile object, read the contents (usually filenames) and return a list. Blank lines and anything after a # comment character are ignored.

```
@files = read_file_list( $listfile );
```

Returns array or array reference depending on calling context. Returns undef if the given file does not exist.

write_file_list

Given an array of file names, open a temp file, write the filenames to it and return the name of the file. The returned object stringifies to the actual filename. It must be returned as an object so that the temp file will be deleted automatically when the variable goes out of scope.

```
$fobject = write_file_list( $Frm->files );
```

Suitable for creating a file to be used for Starlink application group parameters.

Note that overwriting the return value (by, say, string concatenation) runs the destructor which unlinks the temporary file.

write_file_list_inout

Write an input indirection file and an output indirection file using the supplied file suffix.

```
($in, $out, @outfiles) = write_file_list_inout( $Frm, "_al" );
```

The first argument is the frame or group object that will be used for the inout() method and the second argument is the suffix to be supplied to the inout() method. The names of the derived output files are returned in the list. The object is not updated automatically.

If the third (optional) argument is true the output files will be pushed onto the intermediates array associated with the supplied frame/group object. This ensures the files will be cleared up even if they are not output from a primitive. If istmp is true, the output files are not returned to the caller.

```
($in, $out) = write_file_list_inout( $Frm, "_al", 1);
```

hardlink

Create a hard link from an input file to an output file.

```
$status = hardlink( $file, $link );
```

If $out exists, then it will be overwritten by the link.

Returns 1 if successful, 0 otherwise, and puts the error code into $!.

oractime2mjd

Convert the standard ORACTIME format date (YYYYMMDD.frac) to a modified Julian day.

```
  $mjd = oractime2mjd( $oractime );
```

oractime2iso

Convert the standard ORACTIME format date (YYYYMMDD.frac) to an ISO format string.

```
  $iso = oractime2iso( $oractime );
```

oractime2dt

Convert the standard ORACTIME format date (YYYYMMDD.frac) to a DateTime object.

```
  $dt = oractime2dt( $oractime );
```

filter_quoted_string

Prepare a string for use in Starlink command lines, such that the string may be passed to a character parameter without losing any quotation marks within the string. This will typically be passed to an NDF character component.

Performs this by doubling any quotation marks present within the string, and escaping these too, if necessary. This should be used where the string is a priori unknown and may contain quotes such as those to represent arcseconds and arcminutes, or a possessive like "Barnard’s Loop". A common example is a user-defined object name.

```
    $filtered = filter_quoted_string($string, $single);
```

`$single` should be true if the string has been wrapped in single quotes. For example,

```
    my $title = '$object offset by 10"';
```

Otherwise it is assumed to have been enclosed in double quotes. For example,

```
    my $title = "Halley's Comet";
```

SEE ALSO

POSIX, *List::Util*, *Math::Trig*

COPYRIGHT

C.18 ORAC::Group

Base class for dealing with observation groups in ORAC-DR

SYNOPSIS

```
use ORAC::Group;

$Grp = new ORAC::Group("groupid");

$Grp->file("Group_file_name");
$group_name = $Grp->name;
$Grp->push($frame);
$total_in_group = $Grp->num;
$frame3 = $Grp->frame(2);
@good_members = $Grp->members;
```

DESCRIPTION

This module provides the basic methods available to all **ORAC::Group** objects. This class should be used when storing information relating to a group of observations processed in the **ORAC-DR** data reduction pipeline.

Groups are composed of frame objects (**ORAC::Frame**) or objects that can perform those methods.

PUBLIC METHODS

The following methods are available in this class.

Constructors The following constructors are available:

new

Create a new instance of a **ORAC::Group** object. This method takes an optional argument containing the identifier of the new group. The object identifier is returned.

```
$Grp = new ORAC::Group;
$Grp = new ORAC::Group("group_id");

$Grp = new ORAC::Group("group_id", $filename );
```

The base class constructor should be invoked by sub-class constructors. If this method is called with the last argument as a reference to a hash it is assumed that this hash contains extra configuration information ('instance' information) supplied by sub-classes.

configure

Initialise the object.

subgrp

Method to return a new group (ie a sub-group of the existing group) that contains all members of the main group matching certain header values.

Arguments is a hash that is used for comparison with each frame.

```
  $subgrp = $Grp->subgrp(NAME => 'CRL618', CHOP=> 60.0);
```

The new subgrp is blessed into the same class as $Grp. All header information (hdr() and uhdr()) is copied from the main group to the sub-group.

This method is generally used where access to members of the group by some search criterion is required.

It is possible that the returned group will contain no members....

If the value looks like a number a numeric comparison will be performed. Else a string comparison is used.

subgrps

Returns frames grouped by the supplied header keys. A frame can not belong to more than one sub group created by this method:

```
   @grps = $Grp->subgrps(@keys);
```

The groups in @grps are blessed into the same class as $Grp. For example, if @keys = ('MODE','CHOP') then you can gurantee that the members of each sub group will have the same values for MODE and CHOP.

All header information from the main group is copied to the sub groups.

If a key is not present in the headers then all the frames will be returned in a single subgrp (since that group guarantees that the specified header item is not different - it simply is not there).

The order of the subgroups will match the position of the first member in the original group.

Accessor methods The following methods are available for accessing the 'instance' data.

allmembers

Set or retrieve the array containing the current group membership.

```
    $Grp->allmembers(@frames);
    @frames = $Grp->allmembers;
```

The setting function of this routine should only be used if you know what you are doing (since it completely changes the group membership). If setting the contents, the check_membership() method is run automatically so that the list of valid members can remain synchronized.

All group members are returned regardless of the state of each member. Use the members() method to return only valid members.

If called in a scalar context, a reference to an array is returned rather than the array.

```
$ref = $Grp->allmembers;
$first = $Grp->allmembers->[0];
```

Do not use this array reference to change the contents of the array directly unless the check_membership() method is run immediately afterwards. The check_membership() method is responsible for checking the state of each member and copying them to the members() array.

purge_members

Removes all frames from the group. Can be used to reduce the memory footprint of the pipeline when a recipe is designed such that it never needs to go back to the original members. An optional argument may be given to indicate that the last Frame object should be retained.

```
$Grp->purge_members;
```

```
$Grp->purge_members(1);
```

badobs_index

Return (or set) the index object associate with the bad observation index file. A index of class **ORAC::Index::Extern** is used since this index is modified by an external user/program.

The index is created automatically the first time this method is invoked.

coadds

Return (or set) the array containing the list of frame numbers that have been coadded into the current group. This is not necessarily the same as the return of the membernumbers() method since that can return numbers for all the members of the group even if the full coaddition has not taken place or the pipeline has been resumed partway through a coaddition (in which case the coadds array will contain more numbers than are in the group).

```
@coadds = $Grp->coadds;
$coaddref = $Grp->coadds;
$Grp->coadds(@numbers);
```

Returns an array reference in a scalar context, an array in an array context.

The contents of this array are not automatically written to the group file when changed, see the coaddspush() or coaddswrite() methods for further information on object persistence. The array is simply meant as a storage area for the pipeline.

filesuffix

Set or retrieve the filename suffix associated with the reduced group.

```
$Grp->filesuffix(".sdf");
$group_file = $Grp->filesuffix;
```

fixedpart

Set or retrieve the part of the group filename that does not change between invocation. The output filename can be derived using this.

```
    $Grp->fixedpart("rg");
    $prefix = $Grp->fixedpart;
```

is_frame

Whether or not the current object is an ORAC::Frame object.

```
  $is_frame = $self->is_frame;
```

Returns 0 for Group objects.

members

Retrieve the array containing the valid objects within the group

```
    @frames = $Grp->members;
```

This is the safest way to access the group members since it only returns valid frames to the caller.

Use the allmembers() method to return all members of the group regardless of the state of the individual frames.

Group membership should not be set using ths method since that may lead to a situation where the actual membership of the group does not match the selected membership. [Valid group membership should only be set from within this class].

If called in a scalar context, a reference to an array is returned rather than the array.

```
  $first = $Grp->members->[0];
```

name

Retrieve or set the name of the group.

```
    $Grp->name("group_name");
    $group_name = $Grp->name;
```

If no name is set but is retrieved, a name string will be set automatically based on the first frame in the group.

groupid

Set or retrieve the group identifier. This will be the string derived from the OBSERVATION_GROUP header of the first input frame object.

```
    $Grp->groupid("group_id");
    $group_id = $Grp->groupid;
```

General methods The following methods are provided for manipulating **ORAC::Group** objects:

check_membership

Check whether any of the members of the group have been marked for removal from the group. The valid group members are copied to a new array and can be retrieved by the members() method. Note that all group methods use the list of valid group members.

This routine is automatically run whenever the group membership is updated (via the push() or allmembers() methods. This may cause too high an overhead with push() in, for example, the subgrps method).

This method works by looking in a text file created by the observer in $ORAC_DATA_OUT called index.badobs. This file contains a list of numbers (two per line) relating to observations that should be turned off. The first number is the UT date (YYYYMMDD) and the second number is the observation number. This is necessary so that ORAC_DATA_OUT can be reused for a different UT date without worrying about the index file file turning off incorrect observations.

The UT and observation number are compared with each member of the group (the full list of members - see allmembers()). For each group member, the following test is performed to test for validity. First it is queried to check whether it is in a good state (ie has been processed successfully). A frame will be marked as bad if the recipe fails to execute successfully. If the frame is good (from the pipeline viewpoint) the UT date and observation number is then compared with the entries in the index file. If a match can **NOT** be found the frame is considered to be valid and is copied to the list of valid group members (see the members() method).

The format of the index file should be of the form:

```
24 19980716
27 19980716
43 19980815
...
```

This method returns a list, the first element being an array reference to the good members, and the second being an array reference to the bad (or removed) members. Hidden members (negative isgood()) will not be returned.

coaddspush

Used to push observation numbers onto the coadds() array. Automatically runs coaddswrite() to update to sync the file contents with the coadds() array.

```
$Grp->coaddspush(@numbers);
```

coaddspresent

Compares the contents of the coadds() array with the supplied (single) argument. Returns true if the argument is present in the coadds() array, false otherwise. Also, returns false if no arguments are supplied or if the argument is undef.

```
$present = $Grp->coaddspresent($number);
```

coaddsread

Reads the coadds() information from the current group file and stores it in the group using the coadds() method. Should return ORAC__OK if the coadds information was read successfully, else returns ORAC__ERROR.

This is an abstract method and should be defined by a subclass.

coaddswrite

Method to write the contents of the coadds() array to the current group file. Should return ORAC__OK if the coadds information was written successfully, else returns ORAC__ERROR.

If coadds() contains no entries, all coadds information is removed from the group file if present.

This is an abstract method and should be defined by a subclass.

erase

Erases the group file from disk.

```
 $Grp->erase;
```

Returns ORAC__OK if successful, ORAC__ERROR otherwise.

file_exists

Method to determine whether the group file() exists on disk or not. Returns true if the file is there, false otherwise. Effectively equivalent to using `-e` but allows for the possibility that the information stored in file() does not directly relate to the file as stored on disk (e.g. a .sdf extension).

file_from_bits

Method to return the group filename derived from a fixed variable part (eg UT) and a group designator (usually obs number). The full filename is returned (including suffix).

```
$file = $Grp->file_from_bits("UT","num");
```

For the base class the return string is of the format

```
fixedpart . prefix . '_' . number . suffix
```

For example, with IRCAM using a UT date of 980104 and observation number 25 the returned string would be 'rg980104_25.sdf'.

file_from_name

Method to return the group filename using the name() method of the current Group object. The full filename is returned (including suffix).

```
$file = $Grp->file_from_name;
```

If the name() has not been initialized, then this method returns undef. Otherwise, it removes all hashes in the string returned by name() and replaces them with underscores.

The fixedpart() and filesuffix() are prepended and appended, respectively.

frame

Retrieve the nth frame of the group. Counting starts at 0 as for a standard perl array.

```
$Frm = $Grp->frame(2);
```

This is equivalent to

```
$Frm = $Grp->members->[2];
```

A second argument can be used to set the nth frame.

```
$Grp->frame(3, $Frm);
```

Note that this replaces the nth frame in the list of valid members and also replaces the equivalent frame in the list of all members of the group. This is done since the nth valid member is not necessarily the nth group member. If the supplied position is greater than the current number of members the supplied frame is simply pushed onto the array. Remember that just because a frame has been inserted into the group does not necessarily mean that it will be a valid member (check_membership() will be run when setting any member of the group). If the current frame at the specified position can not be found in allmembers() the supplied frame is pushed onto allmembers() and membership is re-checked.

members_inout

Method to return the current filenames for each frame in the group (similar to the membernames() method) and a set of output names for each file. This is achieved by calling the inout() method for each frame in turn. This will fail if the members of the group do not possess the inout() method.

This method can take two arguments: the new suffix and, optionally, the file number to use (see the inout() documentation for **ORAC::Frame**). References to two arrays are returned when called in an array context; returns the output array ref when called from a scalar context

```
($inref, $outref) = $Grp->members_inout("suffix");
($inref, $outref) = $Grp->members_inout("suffix",2);
$outref= $Grp->members_inout("suffix");
```

firstmember

Method to determine whether the supplied argument matches the first member of the group. Returns a 1 if it is the first member and a zero otherwise.

```
$isfirst = $Grp->firstmember( $Frm );
```

lastmember

Method to determine whether the supplied argument matches the last good member of the group. Returns a 1 if it is the last member and a zero otherwise.

```
$islast = $Grp->lastmember($Frm);
```

If there are no good members in the group, returns false.

lastallmembers

Method to determine whether the supplied argument matches the last member of the group. This method differs from lastmember() in that this method does not care about badness, i.e., it looks at the allmembers() list instead of the members() list.

Returns true if the supplied argument is the last member, and false otherwise.

memberindex

Given a frame, determines what position this frame has in the group. This is useful in Batch mode processing where the groups are pre-populated.

```
$index = $Grp->memberindex( $Frm );
```

Index starts counting at 0 (see the `frame` method) and refers only to valid members rather than all members. If the frame is not in the group, returns undef.

membernames

Return a list of all the files associated with the group. This is achieved by invoking the file() method for each object stored in the Members array. For this to work each member must be an object capable of invoking the file() method (e.g. **ORAC::Frame**). Currently the routine does not check to make sure this is possible - the program will die if you try to use a SCALAR member.

If an argument list is given the file names for each member of the group are updated. This will only be attempted if the number of arguments given matches the number of members in the group.

```
$Grp->membernames(@newnames);
@names = $Grp->membernames;
```

Only the first file from each frame object is returned.

membernumbers

Return a list of all the observation numbers associated with the group. This is achieved by invoking the number() method for each object stored in the Members array. For this to work each member must be an object capable of invoking numbers() (e.g. **ORAC::Frame**). Currently the routine does not check to make sure this is possible - the program will die if you try to use a SCALAR member.

```
@numbers = $Grp->membernumbers;
```

membertagset

Set the tag in each of the members.

```
$Grp->membertagset( 'TAG' );
```

Runs the `tagset` method on each of the member frames.

Note that $Grp->tagset() is used for the group filenames not the input files from Frame objects.

membertagretrieve

Run the `tagretrieve()` method for each of the members.

```
$Grp->membertagretrieve
```

Note that $Grp->tagretrieve() is used for the group filenames not the input files from Frame objects.

num

Return the number of frames in a group minus one. This is identical to the $# construct.

```
$number_of_frames = $Grp->num;
```

push

Method to push an observation into the group. Multiple observations can be pushed on at once (see *perl* "push()" command).

```
$Grp->push("observation2");
$Grp->push(@obs);
```

There are no return arguments.

template

Method to change all the current filenames in the group so that they match the supplied template. This method invokes the template() method for each member of the group.

```
$Grp->template("filename_template");
```

A second argument can be specified to modify the specified frame number rather than simply the first (see the template() method in **ORAC::Frame** for more details):

```
$Grp->template($template,2);
```

There are no return arguments. The intelligence for this method resides in the individual frame objects.

updateout

This method updates the current filename of each member of the group when supplied with a suffix (and optionally, a file number -- see the inout() method in **ORAC::Frame** for more information). The inout() method (of the individual frame) is invoked for each member to generate the output name.

```
$Grp->updateout("suffix");
$Grp->updateout("suffix", 5);
```

This can be used to update the member filenames after an operation has been applied to every file in the group. Alternatively the membernames() method can be invoked with the output of the inout() method.

DISPLAY COMPATIBILITY

These methods are provided for compatibility with the ORAC display system.

gui_id

Returns the identification string that is used to compare the current frame with the frames selected for display in the display definition file.

In the default case, this method returns everything after the last suffix stored in file().

In some derived implementation of this method an argument may be used so that multiple IDs can be extracted from objects that contain more than one output file per observation.

SEE ALSO

ORAC::Frame

COPYRIGHT

C.19 ORAC::Group::NDF

Class for dealing with groups based on NDF files

SYNOPSIS

```
use ORAC::Group::NDF

$Grp = new ORAC::Group::NDF;
```

DESCRIPTION

This class rovides implementations of the methods that require knowledge of the NDF file format rather than generic methods or methods that require knowledge of a specific instrument. In general, the specific instrument sub-classes will inherit from the file type (which inherits from ORAC::Group) rather than directly from ORAC::Group. For JCMT and UKIRT the group files are based on NDFs and inherit from this class.

The format specific sub-classes do not contain constructors; they should be defined in either the base class or the instrument specific sub-class.

PUBLIC METHODS

The following methods are modified from the base class versions.

General Methods

coaddsread

Method to read the COADDS information from the group file. If the Group file exists, the file is opened and the *.ORAC* extension is located. The .COADDS component (ie groupfile.MORE.ORAC.COADDS) is then opened as _INTEGER and the contents are stored in the group using the coadds() method. If a .MORE.ORAC.COADDS component can not be found (e.g. because the file or component do not exist), the routine returns ORAC__ERROR, else returns ORAC__OK.

```
$Grp->coaddsread;
```

There are no arguments.

coaddswrite

Writes the current contents of coadds() into the current group file(). Returns ORAC__OK if the coadds information was written successfully, else returns ORAC__ERROR.

```
$Grp->coaddswrite;
```

There are no arguments. The information is written to a .ORAC.COADDS component in the Group file. If coadds() contains no entries, all coadds information is removed from the group file if present (and good status is returned). A .ORAC extension is always made if one does not exist and the file is present.

REQUIREMENTS

This module requires the *NDF* module.

SEE ALSO

ORAC::Group, ORAC::BaseNDF

COPYRIGHT

C.20 ORAC::Group::UKIRT

Base class for dealing with groups from UKIRT instruments

SYNOPSIS

```
  use ORAC::Group::UKIRT;

  $Grp = new ORAC::Group::UKIRT;
```

DESCRIPTION

This class provides UKIRT specific methods for handling groups.

COPYRIGHT

C.21 ORAC::Group::WFCAM

Class for dealing with WFCAM observation groups in ORAC-DR

SYNOPSIS

```
  use ORAC::Group::WFCAM;

  $Grp = new ORAC::Group::WFCAM("group1");
  $Grp->file("group_file")
  $Grp->readhdr;
  $value = $Grp->hdr("KEYWORD");
```

DESCRIPTION

This module provides methods for handling group objects that are specific to WFCAM. It provides a class derived from **ORAC::Group::UFTI**. All the methods available to ORAC::Group objects are available to **ORAC::Group::WFCAM** objects.

PUBLIC METHODS

The following methods are available in this class in addition to those available from ORAC::Group.

Constructor

new

Create a new instance of a **ORAC::Group::WFCAM** object. This method takes an optional argument containing the name of the new group. The object identifier is returned.

```
$Grp = new ORAC::Group::WFCAM;
$Grp = new ORAC::Group::WFCAM("group_name");
```

This method calls the base class constructor but initialises the group with a file suffix of '.sdf' and a fixed part of 'g'.

General Methods

SEE ALSO

ORAC::Group, *ORAC::Group::NDF*

COPYRIGHT

C.22 ORAC::Group::ACSIS

ACSIS class for dealing with observation groups in ORAC-DR

SYNOPSIS

```
use ORAC::Group;

$Grp = new ORAC::Group::ACSIS("group1");
$Grp->file("group_file")
$Grp->readhdr;
$value = $Grp->hdr("KEYWORD");
```

DESCRIPTION

This module provides methods for handling group objects that are specific to ACSIS. It provides a class derived from **ORAC::Group::NDF**. All the methods available to **ORAC::Group** objects are available to **ORAC::Group::ACSIS** objects.

PUBLIC METHODS

The following methods are available in this class in addition to those available from **ORAC::Group**.

Constructor

new

Create a new instance of an **ORAC::Group::ACSIS** object. This method takes an optional argument containing the name of the new group. The object identifier is returned.

```
$Grp = new ORAC::Group::ACSIS;
$Grp = new ORAC::Group::ACSIS("group_name");
```

This method calls the base class constructor but initialises the group with a file suffix if ".sdf" and a fixed part of "ga".

General Methods

file_from_bits

Method to return the group filename derived from a fixed variable part (eg UT) and a group designator (usually obs number). The full filename is returned (including suffix).

```
$file = $Grp->file_from_bits("UT","num","extra");
```

For ACSIS the return string is of the format

```
fixedpart . prefix . '_' . number . '_' . extra . suffix
```

inout

Similar to base class except the frame number is appended to the output suffix. The frame number is padded with zeroes to make it three digits long.

SEE ALSO

ORAC::Group::NDF

COPYRIGHT

C.23 ORAC::Group::SCUBA

SCUBA class for dealing with observation groups in ORAC-DR

SYNOPSIS

```
use ORAC::Group::SCUBA;

$Grp = new ORAC::Group::SCUBA("group1");
$Grp->file("group_file")
$Grp->readhdr;
$value = $Grp->hdr("KEYWORD");
```

DESCRIPTION

This module provides methods for handling group objects that are specific to SCUBA. It provides a class derived from **ORAC::Group::NDF**. All the methods available to **ORAC::Group** objects are available to **ORAC::Group::SCUBA** objects. Some additional methods are supplied.

PUBLIC METHODS

The following methods are available in this class in addition to those available from **ORAC::Group**.

Constructor

new

Create a new instance of a **ORAC::Group::SCUBA** object. This method takes an optional argument containing the name of the new group. The object identifier is returned.

```
$Grp = new ORAC::Group::SCUBA;
$Grp = new ORAC::Group::SCUBA("group_name");
```

This method calls the base class constructor but initialises the group with a file suffix of '.sdf' and a fixed part of '_grp_'.

General Methods

file

This is an extension to the default file() method. This method accepts a root name for the group file (independent of sub-instrument) - same as for the base class. If a number is supplied the root name is returned with the appropriate extension relating to the sub-instrument order in the current frame.

The number to sub-instrument conversion uses the last frame in the group to calculate the allowed number of sub-instruments and the order. Note that this may well not be what you want. Use the grpoutsub() method if you know the name of the sub-instrument.

file_from_bits

Method to return the group filename derived from a fixed variable part (eg UT) and a group designator (usually obs number). The full filename is returned (including suffix).

```
  $file = $Grp->file_from_bits("UT","num");
```

Returns file of form UT_grp_00num.sdf

Note that this is the filename before sub-instruments have been taken into account (essentially this is the default root name for file() - the suffix is stripped).

file_from_name

Override the version in the base class since SCUBA group files use a non-standard component order.

Does not use the **name** method.

gui_id

The file identification for comparison with the **ORAC::Display** system. Input argument is the file number (starting from 1).

This routine calculates the current suffix from the group file name base and prepends a string 'gN' signifying that this is a group observation and the Nth frame is requested (N is less than or equal to nfiles()).

The assumption is that file() returns a root name (ie without a sub-instrument designation). This then allows us to create an ID based on number and suffix without having to chop the sub-instrument name off the end.

nfiles

This method returns the number of files currently associated with the group. What this in fact means is that it returns the number of files associated with the last member of the group (since that is how I construct output names in the first place). grpoutsub() method is responsible for converting this number into a filename via the file() method.

inout

Local version of `inout`. This version does not take a numeric argument since that would force the file() method to return the sub-instrument as part of the name.

```
($inroot, $outroot) = $Grp->inout( "reb" );
$outroot = $Grp->inout( "reduced" );
```

files

Returns all the filenames associated with the group.

```
@files = $Grp->files();
```

Unlike the standard implementation this can not be used to update the list of files since that is derived from the root filename and the sub-instruments.

NEW METHODS

This section describes methods that are available to the SCUBA implementation of ORAC::Group.

grpoutsub

Method to determine the group filename associated with the supplied sub-instrument.

This method uses the file() method to determine the group rootname and then tags it by the specified sub-instrument.

```
$file = $Grp->grpoutsub($sub);
```

membernamessub

Return list of file names associated with the specified sub instrument.

```
@names = $Grp->membernamessub($sub)
```

subs

Returns an array containing all the sub instruments present in the group (some frames may only have one sub-instrument)

```
@subs = $Grp->subs;
```

The frames should be able to invoke the subs() method.

SEE ALSO

ORAC::Group

COPYRIGHT

Copyright (C) 1998-2000 Particle Physics and Astronomy Research Council. All Rights Reserved.

C.24 ORAC::Group::SCUBA2

SCUBA2 class for dealing with observation groups in ORAC-DR

SYNOPSIS

```
use ORAC::Group;

$Grp = new ORAC::Group::SCUBA2("group1");
$Grp->file("group_file")
$Grp->readhdr;
$value = $Grp->hdr("KEYWORD");
```

DESCRIPTION

This module provides methods for handling group objects that are specific to SCUBA2. It provides a class derived from **ORAC::Group::NDF**. All the methods available to **ORAC::Group** objects are available to **ORAC::Group::SCUBA2** objects.

PUBLIC METHODS

The following methods are available in this class in addition to those available from **ORAC::Group**.

Constructor

new

Create a new instance of an **ORAC::Group::SCUBA2** object. This method takes an optional argument containing the name of the new group. The object identifier is returned.

```
$Grp = new ORAC::Group::SCUBA2;
$Grp = new ORAC::Group::SCUBA2("group_name");
```

This method calls the base class constructor but initialises the group with a file suffix if ".sdf" and a fixed part of "ga".

file_from_bits

Method to return the group filename derived from a fixed variable part (eg UT), a group designator (usually obs number) and the observing wavelength. The full filename is returned (including suffix).

```
$file = $Grp->file_from_bits("UT","num","wavelen");
```

For SCUBA-2 the return string is of the format

```
fixedpart . prefix . '_' . number . '_' . wavelen . suffix
```

where the number is a 5-digit zero-padded integer and the wavelen is either 850 or 450.

General Methods

frmhdrvals

Returns all the FITS header values associated with a particular keyword used in all of the Frames that are members of this group (disabled frames are not used).

```
@values = $Grp->memberhdrvals( $keyword );
```

Calls the `hdrvals` method in each member and collates the results to remove duplicates. Order is retained.

refimage

Set or retrieve the name of a reference image used to define a coordinate system when mosaicking. The name of the reference image is stored in the Group uhdr as `REFIMAGE_KEY` where `KEY` is either `SKY` of `FPLANE`.

Separate reference images may be defined for on-sky images and focal-plane mosaics as given by the first (optional) argument. A `SKY` reference is assumed if not specified when retrieving a reference. However, the refernce type must be given when storing the name of the reference image.

```
my $skyref = $Grp->refimage;
my $fpref = $Grp->refimage("FPLANE");
$Grp->refimage("FPLANE", $fpref);
$Grp->refimage("SKY", $skyref);
```

sort_by_subarray

Create a new group containing frame objects for the files from each subarray.

```
$subarrayGrp = $Grp->sort_by_subarray;
```

This method should only be used in cases where processing by subarray is required (e.g. flatfield and noise observations).

DISPLAY COMPATIBILITY

These methods are provided for compatibility with the ORAC display system.

gui_id

Returns the identification string that is used to compare the current frame with the frames selected for display in the display definition file.

In the default case, this method returns everything after the last suffix stored in file().

In some derived implementation of this method an argument may be used so that multiple IDs can be extracted from objects that contain more than one output file per observation.

SEE ALSO

ORAC::Group::NDF

COPYRIGHT

C.25 ORAC::Index

Perl routines for manipulating ORAC index files

SYNOPSIS

```
use ORAC::Index;
```

DESCRIPTION

This module provides subs for manipulating ORAC index files. ORAC index files consist of whitespace seperated columns containing information about a particular frame.

In the case of calibration index files, these may also contain rules for determining the suitability of use for these frames. These consist of code that is TRUE or FALSE depending on appropriate header values of the object to be calibrated.

PUBLIC METHODS

The following methods are available in this class.

Constructor

new

Create a new instance of an **ORAC::Index** object.

```
$Index = new ORAC::Index;
$Index = new ORAC::Index($indexfile, $rulesfile);
```

Any arguments are passed to the configure() method.

Accessor Methods

configure

Takes an index file and a rules file and sets up the index object

```
$Index->configure($indexfile, $rulesfile);
```

indexfile

Return (or set) the filename of the index file

```
$file = $Index->indexfile;
$Index->indexfile($file);
```

rulesok

Returns true if we are using a valid set of rules, false if the rules were automatically generated from a read of the index file (and therefore contain no clauses for verification).

indexrulesfile

Return (or set) the filename of the rules file

If the rules file has the magic value of ORAC::Index::NO_RULES a lightweight version of the object will be instantiated that does not do any explicit rules checking. This only works if an index file is being read (since the rules column names will be read from the index file), rather than being freshly created (there will be no columns in the output file!).

rulesref

Returns or sets the reference to the hash containing the rules

indexref

Returns or sets the reference to the hash containing the index

indexkeys

Return all the keys associated with the index file (ie from `indexref` method. These can then be used in conjunction with `indexentry` to obtain the content of the index.

```
@keys = $index->indexkeys;
```

General Methods

slurprules

Sets up the index rules in the object. Croaks if it fails. This converts the index rules file into an internal hash that can be retrieved with the rulesref() method.

slurpindex

Sets up the index data in the object. Croaks if it fails. This converts the index file name into an internal hash that can be retrieved using the indexref() method. There is one optional argument. The supplied argument is used to control the behaviour of the read. If the 'usekey' flag is true the first string in each row (space separated) is used as a key for the index hash.

If 'usekey' is false the key for each row is created automatically. This is useful for indexes where the contents of the index is more important than any particular key.

```
  $index->slurpindex(0); # Auto-generate keys
```

Default behaviour (ie no args) is to read the key from the index file (ie usekey=1).

writeindex

writes out the current state of the index object into the index file

add

adds an entry to an index

```
  $index->add($name,$hashref)
```

append_to_index

Method to force an append of the specified index entry to the the index file on disk.

```
  $Index->append_to_index($name);
```

$name is the name of the key (indexentry) to use to select the index entry to append [cf the indexentry() method].

This method is intended to be called from the add() method to speed up index read/write when appending a new entry. Do not use this method to write a modified entry to the index file (since the original entry will still be on disk)

No return value.

index_to_text

Convert an index entry (in the index hash) to text suitable for writing to an index file. Called by writeindex() and append_to_index()

```
  $text = $Ind->index_to_text($entry);
```

Returns the text string (including the entry name but no carriage return).

ARRAY or HASH references are serialised (although the current output format restricts the use of spaces).

indexentry

Returns a hash containing the key value pairs of the selected index entry.

Input argument is the index entry name (ie the key in the hash that returns the information (in an array).

Returns a hash reference if successful, undef if error.

As an optimization (useful for ORAC::Index::Extern), an index hash can be supplied as the first argument.

verify

verifies a frame (in the form of a hash reference) against a (calibration) index entry (ie by supplying the hash key to the index entry). An optional third argument is available to turn off warning messages -- default is for warning messages to be turned on (true)

```
  $result = $index->verify(indexkey, \%hash, $warn);
```

Returns undef (error), 0 (not suitable), or 1 (suitable)

choosebydt

Chooses the optimal (nearest in time to an observation) calibration frame from the index hash

```
  $calibration = $Index->choosebydt($key, \%header, $warn);
```

Key is the name of the field that should be compared (eg ORACTIME) and %header is the hash containing the header values that are to be compared with the index rules. $warn is an optional third argument that can be used to turn off warning messages from verify (default is to report messages - true).

This method returns the name of the calibration frame closest in time that has met the selection criteria.

If a suitable calibration can not be found an undefined value is returned.

chooseby_positivedt

Chooses the calibration frame closest in time from above by looking in the index file (ie difference between the index file entry and the current frame is positive).

```
  $calibration = $Index->chooseby_positivedt($key, \%header, $warn);
```

Key is the name of the field that should be compared (eg ORACTIME) and %header is the hash containing the header values that are to be compared with the index rules. $warn is an optional third argument that can be used to turn off warning messages from verify (default is to report messages - true).

This method returns the name of the calibration frame closest in time that has met the selection criteria.

This is similar to the choosebydt() method except that only calibrations taken after the current time (read from the header) can be chosen. undef is returned if no suitable calibration frames can be found (eg because we are running on-line and they have not even been taken yet).

chooseby_negativedt

Chooses the calibration frame closest in time from below by looking in the index file (ie delta time between the index entry and the current frame is negative).

```
  $calibration = $Index->chooseby_negativedt($key, \%header, $warn);
```

Key is the name of the field that should be compared (eg ORACTIME) and %header is the hash containing the header values that are to be compared with the index rules. $warn is an optional third argument that can be used to turn off warning messages from verify (default is to report messages - true).

This method returns the name of the calibration frame closest in time that has met the selection criteria.

This is similar to the choosebydt() method except that only calibrations taken before the current time (read from the header) can be chosen. undef is returned if no suitable calibration can be found.

choosebydt_generic

Internal routine for handling calibraion matches using a time difference.

```
  $calibration = $Index->choosebydt_generic(TYPE, $key, \%header, $warn);
```

TYPES can be 'ABS' (chooses the closest calibration in time), 'POSITIVE' (chooses the closest in time from calibrations earlier than the current header) and 'NEGATIVE' (chooses calibrations after the current observation [as described by %header]).

KEY, HEADER and WARN are described in the choosebydt() documentation.

cmp_with_hash

Compares each index entry with the values in the supplied hash (supplied as a hash reference). The key to the first matching index entry is returned. undef is returned if no match could be found.

```
  $key = $index->cmp_with_hash(\%hash);
  $key = $index->cmp_with_hash({ key1 => 'value',
                                 key2 => 'value2'});
```

Use the indexentry() method to convert this key into the actual index entry. Note that warning messages are turned off during the verification stage since we are not interested in failed matches.

Returns 'undef' if no match is found or if no argument is supplied [or that argument itself is undef]

scanindex

Scan the index file for entries that match the supplied constraints. Only string equality constraints are supported. For more complex scans, consider using the rules system directly.

```
  @entries = $index->scanindex( UNITS => 'ARCSEC', FILTER => '850W' );
```

The return entries are not sorted into any particular order.

Regular expression matching is supported by supplying a string beginning and ending with forward slashes (e.g. '/^g/' will match a string starting with 'g').

Matching against the index entry's ID (i.e. the first column in an index) can be done by supplying the hash key ':ID'.

Matching only records more recent than a given ORACTIME can be done by supplying the hash key ':SINCE'.

remove

Remove an item from an index file

_sanity_check

Make sure that the rules and index entry are consistent.

```
 $Idx->_sanity_check( \@calibdata );
```

Takes the entry data as argument (does not try to work out that information itself).

Will die if they are inconsistent.

_num_rules

Returns the number of rules.

_rules_keys

Returns reference to the sorted list of rule keys.

SEE ALSO

ORAC::Index::Extern

COPYRIGHT

C.26 ORAC::LogFile

Routines for generating log files

SYNOPSIS

```
  use ORAC::LogFile;

  $log = new ORAC::LogFile('logfile.dat');
  $log->header(@header);
  $log->addentry(@lines);
  $log->timestamp(1);
```

DESCRIPTION

Provide simple interface to generation of logfiles (eg logging of seeing statistics, photometry results or pointing logs).

PUBLIC METHODS

The following methods are available:

new

Create a new instance of ORAC::LogFile and associate it with the specified log file.

```
$log = new ORAC::LogFile($logfile);
```

If no argument is supplied, the logfile name must be set explcitly by using the logfile() method.

This constructor does not create the logfile itself.

logfile

Return or set the name of the logfile associated with this instance. Usually set directly by the constructor.

```
$logfile = $log->logfile;
$log->logfile($logfile);
```

timestamp

Control whether a timestamp is prepended to each entry written to the logfile. Default is to not print a timestamp.

```
$log->timestamp(1);
$use = $log->timestamp;
```

The timestamp will be in UT.

header

Write header information to the file. Header information is only written if the logfile does not previously exist (since if the file exists already a header is not required). If the logfile does not exist the logfile is created by this method and all arguments written to it. A newline character "\n" is automatically appended to each line.

```
$log->header($line1, $line2);
$log->header(@lines);
```

addentry

Add a log entry. Multiple lines can be supplied (eg as an array). Each line is appended to the logfile (appending a newline "\n" character to each and prepending a timestamp if required).

```
$log->addentry($line);
$log->addentry(@lines);
```

The logfile is closed each time this method is invoked.

COPYRIGHT

Copyright (C) 1998-2001 Particle Physics and Astronomy Research Council. All Rights Reserved.

C.27 ORAC::Loop

Data loops for ORACDR

SYNOPSIS

```
use ORAC::Loop;

$frm = orac_loop_list($class, $utdate, \@list, $skip);

$frm = orac_loop_inf($class, $utdate, \@list);

$frm = orac_loop_wait($class, $utdate, \@list, $skip);

$frm = orac_loop_flag($class, $utdate, \@list, $skip);

$frm = orac_loop_task( $class, \@array, $skip );

$frm = orac_loop_file($class, \@list );
```

DESCRIPTION

This module provides a set of loop handling routines for ORACDR. Each subroutine accepts the same arguments and returns the current observation number (or undef if there was an error or if the loop should be terminated).

A new Frame object is returned of class $class that has been configured for the new file (ie a `$Frm->configure` method has been run)

It is intended that this routine is called inside an infinite while loop with the same @list array. This array is modified by the loop routines so that they can keep track of the 'next' frame number.

If a filename can not be found (eg it doesnt exist or the list has been processed) undef is returned.

The skip flag is used to indicate whether the loop should skip forward if the current observation number can not be found but a higher numbered observation is present. Currently no loops will go back to missing observations if they appear after a higher number (eg observation 10 appears before observation 9!)

LOOP SUBROUTINES

The following loop facilities are available:

orac_loop_list

Takes a list of numbers and returns back a frame object for each number (one frame object per call)

```
$Frm = orac_loop_list($class, $UT, \@array, $noskip);
```

undef is returned on error or when all members of the list have been returned. If the ′skip′ flag is true missing files in the list will be ignored and the next element of the list selected. If ′skip′ is false the loop will abort if the file is not present

orac_loop_inf

Checks for the frame stored in the first element of the supplied array and returns the Frame object if the file exists. The number is incremented such that the next observation is returned next time the routine is called.

```
$Frm = orac_loop_inf($class, $ut, \@array);
```

undef is returned on error or when there are no more data files available.

This loop does not have a facility for skipping files when observations are not present. This behaviour is obtained by combining orac_check_data_dir with the list looping option so that the last observation number can be determined before running the loop. The skip flag is ignored in this loop.

orac_loop_wait

Waits for the specified file to appear in the directory. A timeout of 12 hours is hard-wired in initially -- undef is returned if the timeout is exceeded.

```
$frm = orac_loop_wait($class, $utdate, \@list, $skip);
```

The first member of the array is used to keep track of the current observation number. This element is incremented so that the following observation is returned when the routine is called subsequently. This means that this loop is similar to using the ′-from′ option in conjunction with the ′inf′ loop except that new data is expected.

The loop will return undef (i.e. terminate looping) if the supplied array contains undef in the first entry.

The skip flag is used to indicate whether the loop should skip forward if the current observation number can not be found but a higher numbered observation is present.

If no data can be found, the directory is scanned every few seconds (hard-wired into the routine). A dot is printed to the screen after a specified number of scans (default is 1 dot per scan and one scan every 2 seconds).

orac_loop_flag

Waits for the specified file to appear in the directory by looking for the appearance of the associated flag file. A timeout of 60 minutes is hard-wired in initially -- undef is returned if the timeout is exceeded.

```
  $frm = orac_loop_flag($class, $utdate, \@list, $skip);
```

The first member of the array is used to keep track of the current observation number. This element is incremented so that the following observation is returned when the routine is called subsequently. This means that this loop is similar to using the '-from' option in conjunction with the 'inf' loop except that new data is expected.

The loop will return undef (i.e. terminate looping) if the supplied array contains undef in the first entry.

orac_loop_file

Takes a list of files and returns back frame objects for the files, removing them from the input array.

```
  @Frms = orac_loop_file($class, $ut, \@array, $skip );
```

undef is returned on error or if the list of files is empty.

The UT and skip parameters are ignored.

The input filenames are assumed to come from $ORAC_DATA_IN if they use a relative path.

orac_loop_task

In this scheme ORAC-DR looks for a flag file of a standard name created by a task that is monitoring remote QL parameters and writing the data locally. The flag file is always the same name and contains the files that should be processed by the pipeline immediately.

When the pipeline finds a flag file the file is immediately renamed and the listed files are then "owned" by the pipeline. The QL task monitor will continue to write a new flag file when data arrives and delete the old flag file if the pipeline has not taken ownership. The QL task monitor deletes files that were never harvested. The pipeline should tidy up for itself if it finds the flag file.

```
  $Frm = orac_loop_task( $class, $ut, \@array, $skip );
```

In this looping scheme all except the first argument are ignored.

OTHER EXPORTED SUBROUTINES

orac_check_data_dir

Routine to check the input data directory (ORAC_DATA_IN) for files in order to see whether files exist with a higher number than the supplied number. The routine is supplied with a class name, UT date and current observation number. An additional argument is provided to determine whether data files or flag files should be used for the directory search.

```
$next = orac_check_data_dir($class, $current, $flag);
($next, $high) = orac_check_data_dir($class, $current, $flag);
```

If called in a scalar context, the return argument is the next observation in the sequence. If called in an array context, two arguments are returned: the next observation number and the highest observation number.

undef (or undef,undef) is returned if no higher observations can be found. If it is necessary to check for the existence of current file as well (eg via a data detection loop) then simply decrement the supplied argument by 1.

This routine is used in conjunction with the -from loop (where we dont know the end) and the waiting loops where we are not sure whether new data have been written to disk but missing the next observation.

This routine does NOT look in ORAC_DATA_OUT.

A global variables (@LIST) is used to speed up the sorting by storing a list of observation numbers that have previously been shown to have a lower number than required (NOT YET IMPLEMENTED).

PRIVATE SUBROUTINES

The following subroutines are not exported.

link_and_read

General subroutine for converting ut and number into file and creating a Frame object or multiple frame objects (depending on instrument and mode).

```
@frm = link_and_read($class, $ut, $obsnum, $flag)

@frm = link_and_read($class, $ut, $obsnum, $flag, \@reflist)
```

The five parameters are:

class
Class of Frame object

ut
UT date in YYYYMMDD

obsnum
Observation number

flag
If filename(s) is to come from flag file

reflist
Reference to array of files names that should be excluded from the list of files read from the flag files (if flag files are non-zero). This allows for flag files that can change length during an observation (potentially allowing the pipeline to stop before the full observation is complete but after data files start appearing for the observation). The contents of this array are updated on exit to include the files that were just read. This allows the reference list to be resubmitted to this routine.

Empty list is returned on error.

Returns 1 or more configured frame objects on success.

orac_sleep

Pause the checking for new data files by the specified number of seconds.

```
  $time = orac_sleep($pause);
```

Where $pause is the number of seconds to wait and $time is the number of seconds actually waited. Seconds can be fractional.

If the Tk system is loaded this routine will actually do a Tk event loop for the required number of seconds. This is so that the X screen will be refreshed. Currently the only test is the Tk is loaded, not that we are actually using Tk.....

_find_from_pattern

Given a pattern or string, look in ORAC_DATA_IN and return all the files that are applicable.

```
  @files = _find_from_pattern( $pattern );
```

If the pattern finds .ok files they will be opened. It is assumed that this routine will not be triggered in dynamic flag mode.

_files_there

Return true if all the specified files are present.

```
 if (!_files_there( @files ) {
   ...
 }
```

Returns false is no files are supplied.

_files_nonzero

Return true if all the specified files are present with a size greater than 0 bytes

```
 if (_files_nonzero( @files ) {
   ...
 }
```

Returns false if no files are supplied.

_to_abs_path

Convert a filename(s) relative to ORAC_DATA_IN to an absolute path.

```
  @abs = _to_abs_path( @rel );
```

Does not affect absolute paths.

In scalar context, returns the first path.

_clean_path

Splits path up and resolves "../" entries. This is done because normally if you use a symlink ../ ends up on the other end of the directory symlink.

```
 $clean = _clean_path( $notclean );
```

This may be dangerous....

_convert_and_link

Given the supplied file names, convert and link each file to ORAC_DATA_OUT. If successful returns a list of `ORAC::Frame` objects derived from the input frame.

```
  @frames = _convert_and_link( $Frm, @files );
```

_convert_and_link_nofrm

This is the low level file conversion/linking routine used by `_convert_and_link`.

```
  @converted = _convert_and_link_nofrm( $infmt, $outfmt, @input);
```

Given an input and output format and a list of files, returns the modified files. Returning an empty list indicates an error (but only if @input contained some filenames).

_read_flagfiles

Read the specified flag files and return the contents.

QL Monitor Support Routines to support "task" mode and the "qlgather" task monitor.

_task_flag_file

Name of flag file to monitor in task mode. Includes full path.

```
  $flag = task_flag_file();
```

_is_in_data_out

If the file is present in ORAC_DATA_OUT the name in ORAC_DATA_OUT is returned. Otherwise returns undef. The ORAC_DATA_OUT version of the filename is returned so that a soft link can be detected by the caller even if the file supplied to this routine points to a file outside ORAC_DATA_OUT.

```
  $file_in_data_out = _is_in_data_out( $file );
```

Can include a full path. ORAC_DATA_OUT will be prepended if the file name is not absolute.

clean_flag_file_and_entries

Read the supplied flag file and remove any files that are listed and in $ORAC_DATA_OUT. Then remove the flag file itself.

```
  clean_flag_file_and_entries( $file );
```

COPYRIGHT

C.28 ORAC::Msg::Control::AMS

Control and initialise ADAM messaging from ORAC

SYNOPSIS

```
use ORAC::Msg::Control::AMS;

$ams = new ORAC::Msg::Control::AMS(1);
$ams->init;

$ams->messages(0);
$ams->errors(1);
$ams->timeout(30);
$ams->stderr(\*ERRHANDLE);
$ams->stdout(\*MSGHANDLE);
$ams->paramrep( sub { return "!" } );
```

DESCRIPTION

Methods to initialise the ADAM messaging system (AMS) and control the behaviour.

METHODS

The following methods are available:

Constructor

new

Create a new instance of Starlink::AMS::Init. If a true argument is supplied the messaging system is also initialised via the init() method.

Accessor Methods

messages

Method to set whether standard messages returned from monoliths are printed or not. If set to true the messages are printed else they are ignored.

```
$current = $ams->messages;
$ams->messages(0);
```

Default is to print all messages.

errors

Method to set whether error messages returned from monoliths are printed or not. If set to true the errors are printed else they are ignored.

```
$current = $ams->errors;
$ams->errors(0);
```

Default is to print all messages.

timeout

Set or retrieve the timeout (in seconds) for some of the ADAM messages. Default is 30 seconds.

```
$ams->timeout(10);
$current = $ams->timeout;
```

stderr

Set and retrieve the current filehandle to be used for printing error messages. Default is to use STDERR.

stdout

Set and retrieve the current filehandle to be used for printing normal ADAM messages. Default is to use STDOUT. This can be a tied filehandle (eg one generated by ORAC::Print).

paramrep

Set and retrieve the code reference that will be executed if the parameter system needs to ask for a parameter. Default behaviour is to call a routine that simply prompts the user for the required value. The supplied subroutine should accept three arguments (the parameter name, prompt string and default value) and should return the required value.

```
$self->paramrep(\&mysub);
```

A simple check is made to make sure that the supplied argument is a code reference.

Warning: It is possible to get into an infinite loop if you try to continually return an unacceptable answer.

General Methods

init

Initialises the ADAM messaging system. This routine should always be called before attempting to control I-tasks.

A relay task is spawned in order to test that the messaging system is functioning correctly. The relay itself is not necessary for the non-event loop implementation. If this command hangs then it is likely that the messaging system is not running correctly (eg because the system was shutdown uncleanly.

```
  $ams->init( $preserve );
```

For ORAC-DR the message system directories are set to values that will allow multiple oracdr pipelines to run without interfering with each other.

Scratch files are written to ORACDR_TMP directory if defined, else ORAC_DATA_OUT is used. By default ADAM_USER is set to be a directory in the scratch file directory. This can be overridden by supplying an optional flag.

If `$preserve` is true, ADAM_USER will be left untouched. This enables the pipeline to talk to tasks created by other applications but does mean that the users ADAM_USER may be filled with unwanted temporary files. It also has the added problem that on shutdown the ADAM_USER directory is removed by ORAC-DR, this should not happen if `$preserve` is true but is not currently guaranteed.

CLASS METHODS

require_uniqid

Returns true, indicating that the ADAM "engine" identifiers must be unique in the client each time an engine is launched.

REQUIREMENTS

This module requires the Starlink::AMS::Init module.

SEE ALSO

Starlink::AMS::Init

COPYRIGHT

C.29 ORAC::Msg::Task::ADAM

Load and control ADAM tasks

SYNOPSIS

```
use ORAC::Msg::Task::ADAM;

$kap = new ORAC::Msg::Task::ADAM("kappa","/star/bin/kappa/kappa_mon");

$status           = $kap->obeyw("task", "params");
$status           = $kap->set("task", "param","value");
($status, @values) = $kap->get("task", "param");
($dir, $status)   = $kap->control("default","dir");
$kap->control("par_reset");
$kap->resetpars;
$kap->cwd("dir");
$cwd = $kap->cwd;
```

DESCRIPTION

Provide methods for loading and communicating with ADAM monoliths. This module conforms to the ORAC messaging standard. This is an ORAC interface to the Starlink::AMS::Task module.

By default all tasks loaded by this module will be terminated on exit from perl.

METHODS

The following methods are available:

Constructor

new

Create a new instance of a ORAC::Msg::Task::ADAM object.

```
$obj = new ORAC::Msg::Task::ADAM;
$obj = new ORAC::Msg::Task::ADAM("name_in_message_system","monolith");
$obj = new ORAC::Msg::Task::ADAM("name_in_message_system","monolith"
                                 { TASKTYPE => 'A'} );
```

If supplied with arguments (matching those expected by load()) the specified task will be loaded upon creating the object. If the load() fails then undef is returned (which will not be an object reference).

silent_tasks

Returns or set the list of tasks that should have their output silenced when an obeyw is issued.

```
@silent = $obj->silent_tasks()
$obj->silent_tasks( @silent );
```

The task will be something like "ndftrace" or "stats" and not the monolith name.

General Methods

load

Load a monolith and set up the name in the messaging system. This task is called by the 'new' method.

```
  $status = $obj->load("name","monolith_binary",{ TASKTYPE => 'A' });
```

If the second argument is omitted it is assumed that the binary is already running and can be called by "name".

If a path to a binary with name "name" already exists then the monolith is not loaded.

Options (in the form of a hash reference) can be supplied in order to configure the monolith. Currently supported options are

```
  TASKTYPE  - can be 'A' for A-tasks or 'I' for I-tasks
```

is_silent

Returns a boolean indicating whether the task output should be silenced.

```
$is = $obj->is_silent( "ndftrace" );
```

obeyw

Send an obey to a task and wait for a completion message

```
  $status = $obj->obeyw("action","params");
```

get

Obtain the value of a parameter

```
($status, @values) = $obj->get("task", "param");
```

mget

Get multiple parameter values. This method is a wrapper around the get() method, returning the values in a hash indexed by parameter name. If a parameter has only one value it will be stored in the hash directly, else if multiple values are returned for a parameter a reference to an array will be stored in the return hash.

```
($status, %params) = $obj->mget("task", @params);
```

Status will only be good if all parameter gets return good status. Status will take on the last bad status value but an attempt will be made to get all parameter values even if some return with bad status.

set

Set the value of a parameter

```
$status = $obj->set("task", "param", "newvalue");
```

control

Send CONTROL messages to the monolith. The type of control message is specified via the first argument. Allowed values are:

```
default:  Return or set the current working directory
par_reset: Reset all parameters associated with the monolith.

($current, $status) = )$obj->control("type", "value")
```

"value" is only relevant for the "default" type and is used to specify a new working directory. $current is always returned even if it is undefined.

These commands are synonymous with the cwd() and resetpars() methods.

resetpars

Reset all parameters associated with a monolith

```
$status = $obj->resetpars;
```

cwd

Set and retrieve the current working directory of the monolith

```
($cwd, $status) = $obj->cwd("newdir");
```

contactw

This method will not return unless the monolith can be contacted. It only returns with a timeout. Returns a '1' if we contacted okay and a '0' if we timed out. It will timeout if it takes longer than specified in `ORAC::Msg::ADAM::Control->timeout`.

contact

This method can be used to determine whether the object can contact a monolith. Returns a 1 if we can contact a monolith and a zero if we cant.

pid

Returns process id of forked task. Returns undef if there is no external task.

REQUIREMENTS

This module requires the Starlink::AMS::Task module.

SEE ALSO

Starlink::AMS::Task

COPYRIGHT

C.30 ORAC::Print

ORAC output message printing

SYNOPSIS

```
use ORAC::Print qw/:func/;

orac_print("text",'magenta');
orac_err("error text",'red');

orac_err("error text");
orac_print("some text");
orac_warn("some warning");
orac_notify("some notification");
orac_throw("error text");

$value = orac_read("Prompt");

$prt = new ORAC::Print;
$prt->out("Message","colour");
$prt->notify("OS notification");
$prt->err("Error message");
$prt->war("warning message");
$prt->errcol("red");
$prt->outcol("magenta");
$prt->errbeep(1);
```

```
$prt->logging(1);
$prt->logkey( "_PRIMITIVE_NAME_" );
$prt->out("Log a message" );
@messages = $prt->msglog();
$prt->clearlog();

tie *HANDLE, 'ORAC::Print', $ptr;
```

DESCRIPTION

This module provides commands for printing messages from ORAC software. Commands are provided for printing error messages, warning messages and information messages. The final output location of these messages is controlled by the object configuration.

If the `ORAC::Print::TKMW` variable is set, it is assumed that this is the Tk object referring to the MainWindow, and the `Tk->update()` method is run whenever the `orac_*` commands are executed via the method in the ORAC::Event class. This can be used to keep a Tk log window updating even though no X-events are being processed.

A simplified interface to Term::ReadLine is provided for use with the orac_read command. This can only be used on STDIN/STDOUT and is not object-oriented.

NON-OO INTERFACE

A simplified non-object oriented interface is provided. These routines are exported into the callers namespace by default and are the commands that should be used by primitive writers.

orac_print (text , [colour])

Print the supplied text to the ORAC output device(s) using the (optional) supplied colour.

If the colour is not specified the default value is used (magenta for primtives).

orac_say(text, [colour])

Print the supplied text to the ORAC output device(s) using the (optional) supplied colour. A carriage return is automatically appended to the text to be printed.

orac_warn(text, [colour])

Print the supplied text as a warning message using the supplied colour.

orac_notify(text)

Use the OS notification system (if possible) to report the message.

orac_carp(text, callers, [colour])

Prints the supplied text as a warning message and appends the line number and name of the parent primitive. This information is obtained from the standard $_PRIM_CALLERS_ variable available to each primitive.

orac_err(text, [colour])

Print the supplied text as an error message using the supplied colour.

orac_throw(text, [colour])

Identical to `orac_err` except that an exception is thrown (see `ORAC::Error`) of type `ORAC::Error::FatalErr` immediately after the text message has been printed. Will cause the pipeline to stop processing all data.

orac_term(text, [colour])

Identical to `orac_throw` except that an exception of class `ORAC::Error::TermProcessing` is thrown to force the current recipe to terminate safely. orac_throw will cause the pipeline to abort when called within a recipe.

orac_termerr(text, [colour])

Identical to `orac_term` except that an exception of class `ORAC::Error::TermProcessingErr` is thrown to force the current recipe to terminate safely but with the error state being remembered when the pipeline exits.

orac_debug(text)

Print the supplied text as a debug message using the supplied colour.

orac_read(prompt)

Read a value from standard input. This is simply a layer on top of Term::ReadLine.

```
  $value = orac_read($prompt);
```

There is no Object-oriented version of this routine. It always uses STDIN for input and STDOUT for output.

orac_print_prefix

Set the prefix to be used by `orac_print` in all output.

```
  orac_prefix( "ORAC-DR says:" );
```

orac_printp

As for `orac_print` but includes the prefix that has been specified by using `orac_print_prefix`.

```
 orac_printp( $text, $color );
```

orac_sayp

As for `orac_say` but includes the prefix that has been specified by using `orac_print_prefix`

```
  orac_sayp( $text, $color );
```

orac_warnp

As for `orac_warn` but includes the prefix that has been specified by using `orac_print_prefix`.

```
 orac_warnp( $text, $color );
```

orac_errp

As for `orac_err` but includes the prefix that has been specified by using `orac_print_prefix`.

```
orac_errp( $text, $color );
```

orac_loginfo

Register additional information for logging to the file.

```
orac_loginfo( %info );
```

This information is added to the current set although duplicate keys will overwrite information.

An explicit undef will clear the current information.

```
orac_loginfo( undef );
```

A reference to a hash will force an overwrite of the stored information.

```
orac_loginfo( \%info );
```

To obtain the messages use no arguments:

```
%information = orac_loginfo();
```

orac_clearlog

Clear the message log

```
orac_clearlog();
```

orac_logkey

Set the logging key. Usually the primitive name.

```
orac_logkey( $primitive );
```

orac_logging

Enable or disable logging.

```
orac_logging( 1 );
```

orac_msglog

Returns the logged messages.

```
@messages = orac_msglog();
@messages = orac_msglog( $refepoch );
```

See the msglog() documentation for more information. An undefined reference epoch is equivalent to no reference epoch.

OO INTERFACE

The following methods are available:

Constructors

new()

Object constructor. The object is returned.

Instance Methods

debugmsg

Turns debugging messages on or off. Default is off.

logging

Enables or disables logging of messages. Default is off.

outcol(colour)

Retrieve (or set) the colour currently used for printing output messages.

```
  $col = $prt->outcol;
  $prt->outcol('red');
```

Currently no check is made that the supplied colour is acceptable.

warncol(colour)

Retrieve (or set) the colour currently used for printing warning messages.

```
  $col = $prt->warncol;
  $prt->warncol('red');
```

Currently no check is made that the supplied colour is acceptable.

errcol(colour)

Retrieve (or set) the colour currently used for printing error messages.

```
  $col = $prt->errcol;
  $prt->errcol('red');
```

Currently no check is made that the supplied colour is acceptable.

logkey

String to be associated with output messages. This key will be used when building up the message stack and can be used for grouping purposes.

```
  $prt->logkey( "_IMAGING_HELLO_" );
```

Usually this would reflect the current primitive.

loginfo

Register additional information for logging to the file.

```
  $prt->loginfo( %info );
  $prt->loginfo( KEY => 'value' );
```

This information is added to the current set although duplicate keys will overwrite information.

An undef will delete the key

```
  $prt->loginfo( KEY => undef );
```

An explicit undef will clear the current information.

```
  $prt->loginfo( undef );
```

A reference to a hash will force an overwrite of the stored information.

```
  $prt->loginfo( \%info );
```

To obtain the messages use no arguments:

```
  %information = $prt->loginfo();
```

The information is associated with each change of logkey and so can be updated each time a logkey is updated.

msglog

Array of all logged messages.

```
 @messages = $self->msglog();
```

Each entry is a reference to an array with elements

```
 0 logkey value at time of message
 1 epoch of message
 2 reference to array of messages
 3 reference to hash of log information
```

ie [prim1, epoch, \@msg, \%info], [prim2, epoch2, \@msg, \%info]

Messages will be in epoch order. If an argument is given this will be a reference epoch. Only messages more recent than this will be returned. Only works in list context.

```
 @messages = $self->msglog( $refepoch );
```

An undefined reference epoch is ignored.

prefix

String that is prepended to all messages printed by this class. Default is to have no prefix.

```
$prefix = $prt->prefix;
$prt->prefix('Obs52');
```

outpre

Prefix that is prepended to all strings printed with the out() or say() methods. Default is to have no prefix.

```
$pre = $prt->outpre;
$prt->outpre('ORAC says:');
```

warpre

Prefix that is prepended to all strings printed with the war() or carp() methods. Default is to have the string 'Warning:' prepended.

```
$pre = $prt->warpre;
$prt->warpre('ORAC Warning:');
```

notifypre

Prefix that is prepended to all strings printed with the notify() methods. Default is to have no string prepended.

```
$pre = $prt->notifypre;
$prt->warpre('ORAC-DR:');
```

errpre

Prefix that is prepended to all strings printed with the err() method. Default is to have the string 'Error:' prepended.

```
$pre = $prt->errpre;
$prt->errpre('ORAC Error:');
```

outhdl

Output file handle(s). These are the filehandles that are used to send all output messages. Multiple filehandles can be supplied. Returns an IO::Tee object that can be used as a single filehandle.

```
$Prt->outhdl(\*STDOUT, $fh);
```

```
$fh = $Prt->outhdl;
```

Default is to use STDOUT.

warhdl

Warning output file handle(s). These are the filehandles that are used to print all warning messages. Multiple filehandles can be supplied. Returns an IO::Tee object that can be used as a single filehandle.

```
$Prt->warhdl(\*STDOUT, $fh);

$fh = $Prt->warhdl;
```

Default is to use STDOUT.

errhdl

Error output file handle(s). These are the filehandles that are used to print all error messages. Multiple filehandles can be supplied. Returns an IO::Tee object that can be used as a single filehandle.

```
$Prt->errhdl(\*STDERR, $fh);

$fh = $Prt->errhdl;
```

Default is to use STDERR.

errbeep

Specifies whether the terminal is to beep when an error message is printed. Default is not to beep (false).

```
$dobeep = $Prt->errbeep;
```

debughdl

This specifies the debug file handle. Defaults to STDERR if not defined. Returns an IO::Tee object that can be used as a single filehandle.

Methods

out(text, [col])

Print output messages. By default messages are written to STDOUT. This can be overridden with the outhdl() method.

notify(name, title, text)

Send message to the OS notification system if available. The title is required.

The name (or type) or message must be from an approved list defined as module constants. They are defined at the end of this document.

```
$prt->notify( NOT__INIT, "Initialised pipeline", "Called oracdr_start" );
```

say(text, [col])

Print output messages, appending a carriage return to the text string.

By default messages are written to STDOUT. This can be overridden with the outhdl() method.

war(text, [col])

Print warning messages. Default is to print warnings to STDOUT. This can be overriden with the warhdl() method.

carp(text, callers, [col])

orac_carp(text, callers, [colour])

Prints the supplied text as a warning message and appends the line number and name of the parent primitive. This information is obtained from the standard $_PRIM_CALLERS_ variable available to each primitive.

err(text, [col])

Print error messages. Default is to use STDERR. This can be overriden with the errhdl() method.

throw (text,[colour])

```
$prt->throw("An error message");
```

Same as `err` method except that an exception of type `ORAC::Error::FatalError` is thrown immediately after the error message is printed.

The message itself is part of the exception that is thrown.

term (text,[colour])

```
$prt->term( "An error message" );
```

Similar to orac_throw but can be used in primitives to throw a TermProcessing exception to allow the current recipe to be stopped. orac_throw throws a FatalError exception and so that will completely stop the pipeline.

Not recommended for use outside the recipe execution environment.

termerr (text,[colour])

```
$prt->termerr( "An error message" );
```

Similar to orac_term but can be used in primitives to throw a TermProcessingErr exception to allow the current recipe to be stopped. Unlike orac_term the bad error state will be remembered when the pipeline exits. Can be used when the pipeline recovered from a problem but would like the error to be investigated.

Not recommended for use outside the recipe execution environment.

debug (text)

Prints debug messages to the debug filehandle so long as debugging is turned on.

tk_update ( )

Does an Tk update on the Main Window widget

clearlog

Clear the log. This can be used to reset logged messages when a new recipe begins.

```
  $prt->clearlog();
```

TIED INTERFACE

An ORAC::Print object can also be tied to filehandle using the tie command:

```
tie *HANDLE, 'ORAC::Print', $prt, 'out|war|err';
```

where $prt is an ORAC::Print object. Currently all strings printed to this handle will be redirected to the orac_print command (and will therefore use the output filehandles associated with the most recent ORAC::Print object created). The default color used by the tied handle can be set using the outcol() method of the object associated with the filehandle

```
$prt = new ORAC::Print;
$prt->outcol('clear');
tie *HANDLE, 'ORAC::Print', $prt;
```

will result in all messages printed to HANDLE, being printed with no color codes to STDOUT.

The optional fourth argument to the tie() command can be used to set the default output stream. Allowed values are 'out', 'war' and 'err'. These correspond directly to the orac_print, orac_warn and orac_err commands. Default is to use orac_print for all tied filehandles.

It is not possible to read from this tied filehandle.

SEE ALSO

Term::ANSIColor, *IO::Tee*.

NOTIFICATION TYPES

The following notification types are available

```
NOT__INIT - An initialisation event
NOT__GENERAL  - General unnamed event
NOT__STARTOBS - Detected a new observation
NOT__ENDOBS   - Finished processing an observation
NOT__COMPLETE - Processing has finished
```

COPYRIGHT

C.31 ORAC::TempFile

Generate temporary files for ORAC-DR

SYNOPSIS

```
use ORAC::TempFile;

$temp = new ORAC::TempFile;
$temp = new ORAC::TempFile(0);
$fname = $temp->file;
print {$temp->handle} "Some temporary data";

$temp->handle->close; # Close temporary file

undef $temp;          # Close file and remove it
```

DESCRIPTION

Provide a simplified means of handling temporary files from within ORAC-DR. The temporary file is automatically removed when the object goes out of scope.

The temporary file name can also be used as a temporary name for NDF files. NDF files (extension '.sdf') are automatically deleted in addition to the temporary file created by this class.

PUBLIC METHODS

The following methods are available in this class.

Constructor The following constructors are available:

new

Create a new instance of a **ORAC::TempFile** object.

```
$temp = new ORAC::TempFile;
```

If a false argument is supplied the temporary file name will be allocated (and the file opened) but the file itself will be closed before the new object is returned. This is so that the temporary file name can be passed directly to another process without wanting to write anything to the file yourself (for example if you want to generate a file in an external program and then read the results back into perl).

```
$temp = new ORAC::TempFile(0);
```

Returns 'undef' if the tempfile could not be created. The file is opened for read/write with autoflush set to true. The file should be closed (using the close() method on the object file handle) before sending the file name to an external process (unless a false argument is supplied to the constructor).

```
$temp->handle->close;
```

An argument hash is also supported. These arguments are OPEN and SUFFIX. OPEN is the equivalent of passing in a solitary false argument. SUFFIX appends the given suffix to the temporary filename. For example, to create a temporary file with a closed filehandle and a .txt suffix, one would do:

```
$temp = new ORAC::TempFile( OPEN => 0,
                            SUFFIX => '.txt' );
```

Note that if you pass in an argument hash you cannot have a solitary false argument to return a closed file; if you wish to return a closed file in conjunction with setting a suffix, you must use the OPEN named argument.

Accessor Methods The following methods are available for accessing the 'instance' data.

handle

Return (or set) the file handle associated with the temporary file.

```
print {$tmp->handle} "some information\n";
```

file

Return the file name associated with the temporary file.

```
$name = $tmp->file;
```

The file name is also returned on stringification of the object.

Destructor This section details the object destructor.

DESTROY

The destructor is run when the object goes out of scope or no longer has any references to it. When called, the temporary file is closed and unlinked. If necessary and files of the same name but with a '.sdf' extension are also unlinked. This allows the same class to be used for temporary plain files and temporary NDF files.

No files are removed if the debugging flag ($DEBUG) is set to true (the default is false) or if the ORAC_KEEP environment variable has a value of "temp".

PRIVATE METHODS

The following methods are intended for use inside the module. They are included here so that authors of derived classes are aware of them.

Initialise

This method is used to initialise the object. It is called automatically by the object constructor. It generates a temporary file name and attempts to open it. If the open is not successful the state of the object remains unchanged. In general, this means that the object constructor has failed.

GLOBAL VARIABLES

The following global variables are available. They can be accessed directly or via Class methods of the same name.

- $VERSION

  The current version number of this module.

  ```
  $version = $ORAC::TempFile::VERSION;
  $version = ORAC::TempFile->VERSION;
  ```

- $DEBUG

  Debugging flag. When this flag is set to true the temporary files are not deleted by the object destructor. They can be examined at a later time.

  ```
  $debug = ORAC::TempFile->DEBUG;
  ORAC::TempFile->DEBUG(1);
  $ORAC::TempFile::DEBUG = 0;
  ```

SEE ALSO

IO::File, *File::MkTemp*, tmpnam() in *POSIX*

COPYRIGHT

D Core libraries

This section describes the libraries that are relevant for someone modifying the core.

D.1 ORAC::BaseFile

Shared Base class for Frame and Group classes

SYNOPSIS

```
use base qw/ ORAC::BaseFile /;
```

DESCRIPTION

This class contains methods that are shared by both Frame and Group classes. For example, header and user-header manipulation. File format specific code should not be included (use, for example, `ORAC::BaseNDF`).

PUBLIC METHODS

The following methods are available in this class:

Constructors The following constructors are available:

new

Create a new `ORAC::BaseFile` object. In general this constructor should not be called directly but should be called from a subclass.

```
$file = ORAC::BaseFile->new( $filename );
$file = ORAC::BaseFile->new( \@filenames );
$file = ORAC::BaseFile->new();
```

The filename is optional. Multiple files are supplied as a reference to an array.

The base class constructor should be invoked by sub-class constructors. If this method is called with the last argument as a reference to a hash it is assumed that this hash contains extra configuration information ('instance' information) supplied by sub-classes.

```
$file = ORAC::BaseFile->new( \%internal );
```

Calls the configure method to handle sub-class specific configuration. The file arguments to configure match the arguments to the constructor.

Accessor Methods

allow_header_sync

Whether or not to allow automatic header synchronization when the Frame is updated via either the `file` or `files` method.

```
$Frm->allow_header_sync( 1 );
my $allow = $Frm->allow_header_sync;
```

Defaults to false (0).

file

This method can be used to retrieve or set the file names that are currently associated with the frame. Multiple file names can be stored if required (for example the names associated with different SCUBA sub-instruments).

```
$first_file = $Frm->file;     # First file name
$first_file = $Frm->file(1);  # First file name
$second_file= $Frm->file(2);  # Second file name
$Frm->file(1, value);         # Set the first file name
$Frm->file(value);            # Set the first filename
$Frm->file(10, value);        # Set the tenth file name
```

Note that counting starts at 1 (and not 0 as is normal for Perl arrays) and that the filename can not be an integer (otherwise it will be treated as an array index). Use files() to retrieve all the values in an array context.

If a file has been marked as temporary (ie with the nokeep() method) it is erased (running the erase() method) when the file name is updated.

For example, the second file (file_2) is marked as temporary with `$Frm->nokeep(2,1)`. The next time the filename is updated (`$Frm->file(2,'new_file')`) the current file is erased before the 'new_file' name is stored. The temporary flag is then reset to zero.

If a file number is requested that does not exist, the first member is returned.

Every time the file name is updated, the new file is pushed onto the intermediates() array. This is so that intermediate files can be tidied up when required.

If the first argument is present but not defined the command is treated as if you typed

```
$Frm->file(1, undef);
```

ie the first file is set to undef.

The first time a filename is stored the name will also be stored in `raw()` if no previous entries have been made in `raw`.

If the requested index is the number "0" an exception will be thrown since it is highly unlikely that you wanted a file name called "0".

files

Set or retrieve the array containing the current file names associated with the frame object.

```
$Frm->files(@files);
@files = $Frm->files;

$array_ref = $Frm->files;
```

In a scalar context the array reference is returned. In an array context, the array contents are returned.

The file() method can be used to set or retrieve individual filenames.

The previous files are stored as intermediates (similarly to the `file` method behaviour) and the `nokeep` flag is respected.

Note: It is possible to set and retrieve the array members using the array reference rather than the file() method:

```
  $first = $Frm->files->[0];
```

In this approach, the file numbering starts at 0. The file() method is the recommended way of addressing individual members of this array since the file() method could do extra processing of the string (especially when setting the value, for example the automatic deletion of temporary files).

The first time a filename is stored the name will also be stored in `raw()` if no previous entries have been made in `raw`.

gui_id

Returns the identification string that is used to compare the current frame with the frames selected for display in the display definition file.

Arguments:

```
 number - the file number (as accepted by the file() method)
          Starts counting at 1. If no argument is supplied
          a 1 is assumed.
```

To return the ID associated with the second frame:

```
 $id = $Frm->gui_id(2);
```

If nfiles() equals 1, this method returns everything after the last suffix (using an underscore) from the filename stored in file(1). If nfiles > 1, this method returns everything after the last underscore, prepended with 's$number'. ie if file(2) is test_dk, the ID would be 's2dk'; if file() is test_dk (and nfiles = 1) the ID would be 'dk'. A special case occurs when the suffix is purely a number (ie the entire string matches just "\d+"). In that case the number is translated to a string "num" so the second frame in "c20010108_00024" would return "s2num" and the only frame in "f2001_52" would return "num".

Returns `undef` if the file name is not defined.

nfiles

Number of files associated with the current state of the object and stored in file(). This method lets the caller know whether an observation has generated multiple output files for a single input.

fits

Return (or set) the `Astro::FITS::Header` object associated with the FITS header from the raw data. If you simply want to access individual FITS headers then you probably should be using the `hdr` method.

```
$Frm->fits( $fitshdr );
$fitshdr = $Frm->fits;
```

Translated FITS headers are available using the `uhdr` method.

If no FITS header has been associated with this object, one is automatically created from the `hdr`. This allows the header to be derived from either a FITS object or a normal hash.

format

Data format associated with the current file(). Usually one of 'NDF' or 'FITS'. This format should be recognisable by `ORAC::Convert`.

hdr

This method allows specific entries in the header to be accessed. In general, this header is related to the actual header information stored in the file. The input argument should correspond to the keyword in the header hash.

```
$tel = $Frm->hdr("TELESCOP");
$instrument = $Frm->hdr("INSTRUME");
```

Can also be used to set values in the header. A hash can be used to set multiple values (but does not overwrite other keys).

```
$Grp->hdr("INSTRUME" => "IRCAM");
$Frm->hdr("INSTRUME" => "SCUBA",
          "TELESCOP" => 'JCMT');
```

If no arguments are provided, the reference to the header hash is returned.

```
$Grp->hdr->{INSTRUME} = 'SCUBA';
```

The header can be populated from the file by using the readhdr() method. If a FITS header object has been set via the `fits` method, a new header hash will be created automatically if one does not exist already (via a tie).

If there were two headers in the original FITS header only the last header is returned (in scalar context). All headers are returned in list context.

```
@all = $Frm->hdr("COMMENT");
$last = $Frm->hdr("HISTORY");
```

hdrval

Return the requested header entry, automatically dealing with subheaders. Essentially overrides the standard hdr method for retrieving a header value. Returns undef if no arguments are passed.

```
$value = $Frm->hdrval( "KEYWORD" );
$value = $Frm->hdrval( "KEYWORD", 0 );
```

Both return the values from the first sub-header (index 0) if the value is not present in the primary header.

hdrvals

Return all the different values associated with a single FITS header taken from all sub-headers.

```
@values = $Frm->hdrvals( $keyword );
```

Only unique values are returned. Quickly enables the caller to determine how many distinct states are in the Frame.

inout

Method to return the current input filename and the new output filename given a suffix. For the base class the input filename is chopped at the last underscore and the suffix appended when the name contains at least 2 underscores. The suffix is simply appended if there is only one underscore. This prevents numbers being chopped when the name is of the form ut_num.

Note that this method does not set the new output name in this object. This must still be done by the user.

Returns $in and $out in an array context:

```
($in, $out) = $Frm->inout($suffix);
```

Returns $out in a scalar context:

```
$out = $Frm->inout($suffix);
```

Therefore if in=file_db and suffix=_ff then out would become file_db_ff but if in=file_db_ff and suffix=dk then out would be file_db_dk.

An optional second argument can be used to specify the file number to be used. Default is for this method to process the contents of file(1).

```
($in, $out) = $Frm->inout($suffix, 2);
```

will return the second file name and the name of the new output file derived from this. An explicit undefined value will cause the file() method to be invoked without arguments.

The last suffix is not removed if it consists solely of numbers. This is to prevent truncation of raw data filenames.

intermediates

An array containing all the intermediate file names used during processing. Filenames are pushed onto this array whenever the file() method is used to update the current file information.

```
$Frm->intermediates(@files);
@files = $Frm->intermediates;
push(@{$Frm->intermediates}, $file);
$first = $Frm->intermediates->[0];
```

As for the files() method, returns an array reference when called in a scalar context and an array of file names when called from an array context.

The array does not store information relating to the position of the file in the files() array [ie was it stored as `$Frm->file(1)` or `$Frm->file(2)`]. The order simply reflects the order the files were given to the file() method.

See also the push_intermediates() method.

push_intermediates

Equivalent to

```
push(@{$Frm->intermediates}, @files);
```

but ensures that raw frames are not stored on the intermediates array (do not want to risk deleting raw data).

Returns the number of intermediates that were stored (ie either 0 or the number of file names supplied).

raw

This method returns (or sets) the name of the raw data file(s) associated with this object.

```
$Frm->raw("raw_data");
$filename = $Frm->raw;
```

This method returns the first raw data file if called in scalar context, or a list of all the raw data files if called in list context.

Populated automatically the first time the `files` method is used (or during initial object configuration).

nokeep

Flag used to determine whether the current filename should be erased when the file() method is next used to update the current filename.

```
$Frm->erase($i) if $Frm->nokeep($i);

$Frm->nokeep($i, 1);  # make ith file temporary
$Frm->nokeep($i, 0);  # Make ith file permanent
```

```
$nokeep = $Frm->nokeep($i);
```

The mandatory first argument specifies the file number associated with this flag (same scheme as used by the file() method). An optional second argument can be used to set the flag. 'True' indicates that the file should not be kept, 'false' indicates that the file is permanent.

nokeepArr

Array of flags. Used internally by nokeep() method. Set or retrieve the array containing the flags used by the nokeep() method to determine whether the current filename should be erased when the file() method is next used to update the current filename.

```
    $Frm->nokeepArr(@flags);
    @flags = $Frm->nokeepArr;

    $array_ref = $Frm->nokeepArr;
```

In a scalar context the array reference is returned. In an array context, the array contents are returned.

The nokeep() method can be used to set or retrieve individual flags (the numbering scheme is different).

Note: It is possible to set and retrieve the array members using the array reference rather than the nokeep() method:

```
$first = $Frm->nokeepArr->[0];
```

In this approach, the numbering starts at 0. The nokeep() method is the recommended way of addressing individual members of this array since it could do extra processing of the string.

product

Set or return the "product" of the current File object.

```
$self->product( 'Baselined cube' );
$self->product( 'reduced', 'White-light cube' );
$product = $self->product;
```

A "product" is a description of what the current Frame actually is. For example, in an imaging pipeline this might be "dark-subtracted" or "flat-fielded".

tagexists

Check a given tag against the list of current tags and return true if a match is found.

```
my $tag_exists = $self->tagexists( $tag );
```

The comparison is case sensitive.

tagset

Associate the current filenames with a key (or tag). Once a tag is initialised (it can be any string) the `tagretrieve` method can be used to copy these filenames back into the object so that the `files()` method will use those rather than the current values. This allows the data reduction steps to be "rewound".

```
$Frm->tagset('REBIN');
```

The tag is case insensitive.

tagretrieve

Retrieve the files names from the tag and make them the default filenames for the object.

```
my $status = $Frm->tagretrieve('REBIN');
```

The current filenames are stored in the 'PREVIOUS' tag (unless the PREVIOUS tag is requested).

If the given tag does not exist, then this function returns false. Otherwise returns true.

Automatic header syncing is disabled inside this method.

uhdr

This method allows specific entries in the user-defined header to be accessed. The input argument should correspond to the keyword in the user header hash.

```
$tel = $Grp->uhdr("Telescope");
$instrument = $Frm->uhdr("Instrument");
```

Can also be used to set values in the header. A hash can be used to set multiple values (but does not overwrite other keys).

```
$Grp->uhdr("Instrument" => "IRCAM");
$Frm->uhdr("Instrument" => "SCUBA",
           "Telescope" => 'JCMT');
```

If no arguments are provided, the reference to the header hash is returned.

```
$Frm->uhdr->{Instrument} = 'SCUBA';
```

wcs

This method can be used to retrieve or set the World Coordinate System that is currently associated with the frame. Multiple WCSs can be stored if required (for example the WCSs associated with different ACSIS sub-instruments).

```
$first_wcs = $Frm->wcs;     # First WCS
$first_wcs = $Frm->wcs(1);  # First WCS
$second_wcs= $Frm->wcs(2);  # Second WCS
$Frm->wcs(1, value);         # Set the first WCS
$Frm->wcs(value);            # Set the first WCS
$Frm->wcs(10, wcs);        # Set the tenth WCS
```

Note that counting starts at 1 (and not 0 as is normal for Perl arrays).

If a WCS number is requested that does not exist, the first member is returned.

If the first argument is present but not defined the command is treated as if you typed

```
$Frm->wcs(1, undef);
```

ie the first wcs is set to undef.

General Methods

sync_headers

This method is used to synchronize FITS headers with information stored in e.g. the World Coordinate System.

```
$Frm->sync_headers;
$Frm->sync_headers(1);
```

This method takes one optional parameter, the index of the file to sync headers for. This index starts at 1 instead of 0.

Headers are only synced if the value returned by `allow_header_sync` is true.

header_override_file

Sets the name of the file containing header override information.

_get_header_override

Returns a hash of headers to be overridden by filename.

calc_orac_headers

This method calculates header values that are required by the pipeline by using values stored in the header.

Required ORAC extensions are:

ORACTIME: should be set to a decimal time that can be used for comparing the relative start times of frames. For IRCAM this number is decimal hours, for SCUBA this number is decimal UT days.

ORACUT: This is the UT day of the frame in YYYYMMDD format.

This method should be run after a header is set. Currently the readhdr() method calls this whenever it is updated.

```
%translated = $Frm->calc_orac_headers;
```

This method updates the frame user header and returns a hash containing the new keywords.

configure

This method is used to configure the object. It is invoked automatically if the new() method is invoked with an argument. The file() and readhdr() methods are invoked by this command. A single argument is required (to provide compatibility with subclasses) that either refers to the filename or a reference to an array of filenames. Note that the 2 arg version is only supported by specific subclasses (see documentation).

```
$Frm->configure( $filename );
$Frm->configure( \@files );
```

Multiple raw file names can be provided in the first argument using a reference to an array.

readhdr

A method that is used to read header information from the group file. This method does nothing by default since the base class does not know the format of the file associated with an object.

The calc_orac_headers() method is called automatically.

translate_hdr

Translates an ORAC-DR specific header (such as ORAC_TIME) to the equivalent FITS header(s).

```
%fits = $Frm->translate_hdr( "ORAC_TIME" );
```

In some cases a single ORAC-DR header can be decomposed into multiple FITS headers (for example for SCUBA, ORAC_TIME is a combination of the UTDATE and UTSTART). The hash returned by translate_hdr() will include all the key/value pairs required to generate the ORAC header.

This method will be called automatically to update hdr() values ORAC_ keywords are updated via uhdr().

Returns an empty list if no translation is available.

fullfname

Convert the supplied string to the actual file on disk. This would be a string stored in the files() attribute. Normally a no-op but for the case of Starlink a suffix will be added.

```
$full = $Frm->fullfname( $file );
```

force_product_update

Update the product() value and force the file to receive the update with a sync headers.

```
$Obj->force_product_update( "snr", @filenames );
```

The supplied file names will become the associated with the files() method.

If no files are supplied or if files are supplied and the object has no files associated with it, sync_headers() will be called to force the header updates.

SEE ALSO

ORAC::Frame, *ORAC::Group*

COPYRIGHT

D.2 ORAC::BaseFITS

Base class for FITS file manipulation

SYNOPSIS

```
use base qw/ ORAC::BaseFITS /;
```

DESCRIPTION

This class provides base methods for use by classes that need to manipulate FITS files. For example, `ORAC::Frame::MEF`.

METHODS

General Methods

readhdr

Reads the header from the observation file (the filename is stored in the object). This method sets the header in the object (in general that is done by configure()).

```
$Frm->readhdr;
```

The filename can be supplied if the one stored in the object is not required:

```
$Frm->readhdr($file);
```

but the header in $Frm is over-written. All exisiting header information is lost. The `calc_orac_headers()` method is invoked once the header information is read. If there is an error during the read a reference to an empty hash is returned.

If used as a class method, the filename must be supplied and calc_orac_headers() will not be called.

Returns the `Astro::FITS::Header` object.

parsefname

Return the basename of a FITS file, (that is the name of the file without the .fit, .fits etc. filename extension) as well as the directory, filename suffix and FITS image extension number.

```
($basename,$dir,$suffix,$extn) = $Frm->parsefname($in);
```

The argument is optional. If you supply one, it will extract the basename of the argument stripping off the extension relevant to the object...

NOTES

This class must be in the class hierarchy ahead of the base frame class (`ORAC::BaseFile`) so that the `readhdr` method is picked up correctly.

SEE ALSO

ORAC::Frame::MEF

COPYRIGHT

Copyright (C) 1998-2002 Particle Physics and Astronomy Research Council. All Rights Reserved.

D.3 ORAC::BaseGSD

Base class for NDF file manipulation

SYNOPSIS

```
use base qw/ ORAC::BaseGSD /;
```

DESCRIPTION

This class provides base methods for use by classes that need to manipulate GSD files. For example, `ORAC::Frame::GSD` and `ORAC::Group::GSD`.

METHODS

General Methods

readhdr

Reads the header from the observation file (the filename is stored in the object). This method sets the header in the object (in general that is done by configure()).

```
$Frm->readhdr;
```

The filename can be supplied if the one stored in the object is not required:

```
$Grp->readhdr($file);
```

but the header in $Frm is over-written. All exisiting header information is lost. The `calc_orac_headers()` method is invoked once the header information is read. If there is an error during the read a reference to an empty hash is returned.

Currently this method assumes that the reduced group is stored in GSD format. Only the FITS header is retrieved from the GSD.

If used as a class method, the filename must be supplied and calc_orac_headers() will not be called.

Returns the `Astro::FITS::Header` object.

NOTES

This class must be in the class hierarchy ahead of the base frame class (`ORAC::BaseFile`) so that the `readhdr` method is picked up correctly.

SEE ALSO

ORAC::Frame::GSD, *ORAC::Group::GSD*

COPYRIGHT

D.4 ORAC::BaseNDF

Base class for NDF file manipulation

SYNOPSIS

```
use base qw/ ORAC::BaseNDF /;
```

DESCRIPTION

This class provides base methods for use by classes that need to manipulate NDF files. For example, `ORAC::Frame::NDF` and `ORAC::Group::NDF`.

METHODS

General Methods

collate_headers

This method is used to collect all of the modified FITS headers for a given Frame object and return an updated `Astro::FITS::Header` object to be used by the `sync_headers` method.

```
  my $header = $Frm->collate_headers( $file );
```

Takes one argument, the filename for which the header will be returned.

readhdr

Reads the header from the observation file (the filename is stored in the object). This method sets the header in the object (in general that is done by configure()).

```
    $Frm->readhdr;
```

The filename or filenames can be supplied if the one stored in the object is not required:

```
    $Grp->readhdr($file);
```

but the header in $Frm is over-written. If multiple files are in the frame or if multiple filenames are given the header information will be merged. Merged headers will be stored as subheaders and accessible in the hash interface via $Frm->hdr->{SUBHEADERS}->[n]. By default only data that differs will be in a subheader.

An options hash as first argument can be used to override the default behaviour. Specifically if a single file is given (or stored in the object) but it contains multiple NDF components, the headers can be returned such that the component named HEADER or largest header is the primary and the subheaders are stored by component name.

```
  $Frm->readhdr( { nomerge => 1 }, $filename );
```

The subheaders will then be accessible as $Frm->hdr->{I1} (if the component is called "I1").

All existing header information is lost. The `calc_orac_headers()` method is invoked once the header information is read. If there is an error during the read a reference to an empty hash is returned.

Currently this method assumes that the reduced group is stored in NDF format. Only the FITS header is retrieved from the NDF.

If used as a class method, the filename(s) must be supplied and calc_orac_headers() will not be called.

Returns the FITS header object.

sync_headers

This method is used to synchronize FITS headers with information stored in e.g. the World Coordinate System.

```
$Frm->sync_headers;
$Frm->sync_headers(1);
```

This method takes one optional parameter, the index of the file to sync headers for. This index starts at 1 instead of 0. If a non-number is given it is assumed to be the name of a file.

Headers are only synced if the value returned by `allow_header_sync` is true.

read_wcs

Read the frameset and store the resulting `Starlink::AST` object into the object for later retrieval via the `wcs()` method.

```
$Frm->read_wcs();
```

If a file name or filenames are provided the default behaviour is over-ridden and the frameset is only read for the provided files. The resultant framesets are returned without being stored in the object.

```
$wcs = $Frm->read_wcs( $file );
```

In scalar context the first WCS object is returned.

write_wcs

Write the frameset back into the NDF.

```
$Frm->write_wcs( $frameset );
```

If a file name or filenames are provided the default behaviour is over-ridden and the frameset is only read for the provided files. The resultant framesets are returned without being stored in the object.

```
$wcs = $Frm->write_wcs( $file );
```

In scalar context the first WCS object is returned.

flush_messages

Flush any pending oracdr log messages to the history block of the associated file or files. Each file in the object is modified. Only new history is written to the file.

```
$Frm->flush_messages();
$Grp->flush_messages();
```

Specifying a reference epoch will mean that only a log messages since that epoch will be considered.

```
$Frm->flush_messages( $refepoch );
```

set_app_name

A class method that is used to set the NDF application name from the supplied arguments. Usually triggered automatically on entry to a new primitive.

```
$File->set_app_name( Primitive => $primitive );
```

Call without arguments to reset the name to the ORAC-DR app name and version number.

fullfname

Convert the supplied string to the actual file on disk. This would be a string stored in the files() attribute. HDS components are removed from the name and ".sdf" is added.

```
$full = $Frm->fullfname( $file );
```

PRIVATE METHODS

The following methods are intended for use inside the module. They are included here so that authors of derived classes are aware of them.

stripfname

Method to strip file extensions from the filename string. This method is called by the file() method. We strip all extensions of the form ".sdf", ".sdf.gz" and ".sdf.Z" since Starlink tasks do not require the extension when accessing the file name if Convert has been started.

_find_ndf_children

Given an array of filenames, open each one using HDS and see whether there are any top level NDFs inside.

```
@paths = $frm->_find_ndf_children( @files );
```

If a filename looks like it includes an HDS path (ie a file suffix that is not ".sdf") it will be returned unmodified without being opened.

Options can be used to control behaviour if the first argument is a reference to a hash

```
@paths = $frm->_find_ndf_children( { compnames => 1}, $file );
```

If the "compnames" options is true only the names of component NDFs within an HDS structure will be returned, rather than the filename with paths.

NOTES

This class must be in the class hierarchy ahead of the base frame class (`ORAC::BaseFile`) so that the `readhdr` method is picked up correctly.

SEE ALSO

ORAC::Frame::NDF, *ORAC::Group::NDF*

COPYRIGHT

D.5 ORAC::Basic

Some implementation subroutines

SYNOPSIS

```
use ORAC::Basic;


$Display = orac_setup_display;
orac_exit_normally($message);
orac_exit_abnormally($message);
```

DESCRIPTION

Routines that do not have a home elsewhere.

FUNCTIONS

The following functions are provided:

orac_force_abspath

Force ORAC_DATA_IN and ORAC_DATA_OUT to use an absolute path rather than a relative path. Must be called before pipeline does the first chdir.

```
 orac_force_abspath();
```

Does not canonicalize.

orac_setup_display

Create a new Display object for use by the recipes. This includes the association of this object with a specific display configuration file (*disp.dat*). If a configuration file is not in $ORAC_DATA_OUT one will be copied there from $ORAC_DATA_CAL (or $ORAC_DIR if no file exists in $ORAC_DATA_CAL).

If the $DISPLAY environment variable is not set, the display subsystem will be started but only for use by monitor programs.

The display object is returned.

```
  $Display = orac_setup_display;
```

Hash arguments can control behaviour to indicate master vs monitor behaviour. Options are:

```
  - monitor =>  configure as a monitor (default is to be master) (false)
  - nolocal =>  disable master display, monitor files only.
                Default is to display locally (false)
  - orac_instrument => additional information for display title (where possible)
  - picard_recipe => additional information for display title (where possible)
  - recsuffix => additional information for display title (where possible)
```

Monitor files are always written if a master.

orac_make_title_info

Make informational string to be shown in window titles given the parameters passed as options.

orac_exit_normally

Standard exit handler for oracdr. Should be called instead of `exit()` when the pipeline is complete.

Hash arguments control the behaviour. Allowed keys are:

```
  quiet - Do not print any informational messages to stdout (default is false)
  message - Any string to be printed
  err   - true if the supplied message is an error message
          or if the process should exit with bad status (default is false
          unless error stack is populated)
```

If called with a single argument, it is assumed to be an informational message and is equivalent to using the "message" argument. "err" will default to true if we are called when there are messages in the ORAC::Error stack.

Message is printed using orac_err if we know it is an error message. It will be printed even if "quiet" is true.

orac_exit_abnormally

Exit handler when a problem has been encountered. Normally used a signal handler for SIGINT.

orac_exit_if_error

Wrapper around orac_exit_normally(). Will flush errors, display the error message and exit with bad error status if a defined message is supplied. Useful when handling an error from a try block.

```
orac_exit_if_error( $errtext );
```

Does nothing if there is no error text.

orac_chdir_output_dir

Change to the output directory. If that fails, exit the pipeline.

```
orac_chdir_output_dir();
```

Default output directory is controlled by ORAC_DATA_OUT environment variable.

Takes one argument, a boolean dictating whether or not a check that the data is on an NFS disk should be done. By default this check is done, and if ORAC_DATA_OUT is on an NFS-mounted disk, then the pipeline will exit.

SEE ALSO

ORAC::Core, *ORAC::General*

COPYRIGHT

D.6 ORAC::Convert

Methods for converting data formats

SYNOPSIS

```
use ORAC::Convert

$conv = new ORAC::Convert;
$outfile = $conv->convert($infile, {IN => 'FITS', OUT => 'NDF'});

$outfile = $conv->convert($infile, { OUT => 'NDF'});

$outfile = $conv->fits2ndf($infile);

$conv->infile($infile);
$outfile = $conv->convert;  # uses infile()
```

DESCRIPTION

Provide a system for converting data formats. Currently the only output format supported are:

```
NDF     - simple NDF files
HDS     - HDS containers with .HEADER and .Inn NDFs
FITS    - Simple FITS files
```

The only input formats supported are:

```
NDF     - simple NDF files
FITS    - FITS file
UKIRTIO - UKIRT I/O file
HDS     - HDS containers with .HEADER and .Inn NDFs
          In general this can only be converted to a NDF or FITS
          output file if there is only one data frame in the container.
GMEF    - Gemini Multi-Extension FITS.
INGMEF  - Isaac Newton Group Multi-Extension FITS.
```

In many cases the NDF format is used as the intermediate format for all conversions (should probably use PDLs as the intermediate format....)

Uses the Starlink CONVERT package (via monoliths) where necessary.

Can be used to convert from instrument specific NDF files (eg multi-frame CGS4 data or I- and O- frames for IRCAM) to HDS formats usable by the pipeline (either as HDS containers or NDFs with combined I and O information).

The output filename is always related to the input filename (usually simply with a change of suffix).

METHODS

The following methods are provided:

Constructors

new

Object constructor. Should always be used before initiating a conversion.

```
$Cvt = new ORAC::Convert;
```

Returns undef if there was an error creating the object. No arguments are required.

Accessor Methods The following methods are available for accessing the 'instance' data.

engine_launch_object

Returns the `ORAC::Msg::EngineLaunch` object that can be used to launch algorithm engines as required by the particular conversion.

```
 $messys = $self->messys_launch_object;
```

infile

Method for storing or retreiving the current input filename. Used by default if omitted from convert() methods.

```
  $infile = $Cvt->infile;
```

overwrite

Method for storing or retreiving the flag governing whether a file should be overwritten if it already exists.

If false, the file will be converted regardless.

General Methods

convert

Convert a file to the format specified by options.

```
  ($infile, $outfile) = $Cvt->convert;
  @files = $Cvt->convert($oldfile, { IN => 'FITS', OUT => 'NDF' });
```

File is optional - uses infile() to retrieve the name if not specified. The options hash is optional (assumed to be last argument). If not specified the input format will be guessed and the output format will be set to NDF.

Recogised keywords in the hash are:

```
  IN  => input format (NDF, UKIRTio or FITS)
  OUT => desired output format (NDF or HDS)
```

If 'IN' is not specified it will try to derive the format from name.

The output format is set to NDF if non-specified.

Returns a list containing the input filename and output filename. Neither of these filenames has any directory structure removed.

Output filename is written to the current working directory of the CONVERT monoliths (defaults to the CWD of the program when the monoliths were launched - no attempt is made to correct the CWD of the monoliths before conversion).

Will return an undefined output file if the conversion failed.

guessformat

Given 'name' try to guess data format.

```
$format = $Cvt->guessformat("test.sdf");
```

If no name is supplied, infile() is used to retrieve the current filename.

mon

Returns the algorithm engine object, launching it if required.

```
$object = $Cvt->mon($name);
```

Returns undef if a monolith can not be contacted or fails to start. This is launched using `ORAC::Msg::LaunchEngine`.

Data Conversion Methods

fits2ndf

Convert a fits file to an NDF. Returns the output name.

```
$newfile = $Cvt->fits2ndf;
```

Retrieves the input filename from the object via the infile() method.

ndf2fits

Convert an NDF file to a FITS file.

hds2mef

Convert a HDS file into a multi-extension FITS file

gmef2hds

Convert a GEMINI multi-extension FITS file to an HDS container

ingmef2hds

Convert an ING format Multi-Extension FITS file into an HDS container.

UKIRTio2hds

Converts observations that are taken as a header file plus multiple NDFs into a single HDS container that contains a .HEADER NDF and .Inn NDFs for each of the nn data files. This is the scheme used for IRCAM and CGS4 data at UKIRT.

```
$hdsfile = $Cvt->UKIRTio2hds;
```

This routine assumes the old UKIRT data acquisition system (at least for IRCAM and CGS4) is generating the data files. The name of the header file (aka the O-file) must be stored in the object (via the infile() method) before running this method. The I files are assumed to be in the directory `../idir` relative to the header file with a starting character of 'i' rather than 'o' and are multiple files with suffixes of '_1', '_2' etc. The new output file is named 'cYYYYMMDD_NNNNN' where the date is retrieved from the IDATE header keyword and observation number from the OBSNUM header keyword.

Returns undef on error.

hds2ndf

Converts frames taken as HDS container files (container file with .HEADER and .I1) to a simple NDF file. This method only works for the first frame (.I1).

```
$ndf = $Cvt->hds2ndf;
```

If the input HDS has a .I1 component with FITS headers, then the resulting NDF file has the FITS headers from both the .HEADER and the .I1 components merged. Otherwise, the resulting NDF has the FITS headers from just the .HEADER component. No warning is given if more than one component exists (all higher numbers are ignored).

SEE ALSO

The Starlink CONVERT package.

COPYRIGHT

D.7 ORAC::Core

Core routines for data pipelining

SYNOPSIS

```
use ORAC::Core;

orac_process_frame($CURRENT_RECIPE, $PRIMITIVE_LIST, $opt_showcurrent,
                   $Frm, $Grp, $Cal,\%Mon,$OverRecipe, $instrument);
```

```
orac_store_frm_in_correct_grp($Frm, $GrpType, $GrpHash, $GrpArr, $ut);

orac_print_configuration( $opt_debug, $opt_showcurrent, $log_options,
                          $win_str, \$STATUS_TEXT  );

orac_message_launch( $opt_nomsgtmp, $opt_verbose );

orac_start_algorithm_engines( $opt_noeng, $InstObj );

orac_start_display( $nodisplay );

orac_calib_override( $opt_calib, $calclass );

orac_parse_files( $opt_files );

orac_process_argument_list( $opt_from, $opt_to, $opt_skip, $opt_list,
                            $frameclass );

orac_main_data_loop( $opt_batch, $opt_ut, $opt_resume, $opt_skip,
                     $opt_debug, $recsuffix, $grptrans,
                     $loop, $frameclass, $groupclass,
                     $instrument, $Mon, $Cal, \@obs, $Display, $orac_prt,
                     $ORAC_MESSAGE, \$STATUS_TEXT, $PRIMITIVE_LIST,
                     $Override_Recipe );
```

DESCRIPTION

This module contains the core routines that actually handle the data processing. Routines are provided for constructing groups and for processing those groups, along with routines to do the inital pipeline configuration and algorithm engine startup.

SUBROUTINES

The following subroutines are available:

orac_store_frm_in_correct_grp

Stores the supplied frame into a Grp (usually specified in the Frame), creating a new Group object if necessary. The Group objects are stored in a hash (reference supplied) and, optionally, an array (unless undef). This is so that Groups can be retrieved in the order in which they were created. The GrpType specifies the type of Group that should be created (eg **ORAC::Group::UFTI**, **ORAC::Group::JCMT** etc). The UT is supplied purely so that the Group can be named (using the file_from_bits() method).

```
orac_store_frm_in_correct_grp($Frm, $GrpType, \%Groups, \@Groups,
      $ut, $resume, $transient);
orac_store_frm_in_correct_grp($Frm, $GrpType, \%Groups, undef,
      $ut, $resume, $transient);
```

The resume flag is used to determine the behaviour of the group when it is first created. If resume is false, any existing Group file is removed before proceeding; if it is true, the Group file is retained and any coadd information is read using the coaddsread() Group method.

The transient argument controls whether more than one group can be created. If transient is 1 only a single group is stored in %Groups, although multiple may be created during processing. If transient is -1 then only one group is ever created, and every Frame object goes into that group.

The current Grp (ie the Group associated with the supplied Frm) is returned.

orac_process_frame

This is the core **ORAC-DR** pipeline processing routine. It processes the supplied frame object that belongs to the group object, using the supplied calibration object. The instrument name and default recipe are required for recipe/primitive reading since recipes and primitives are stored in instrument specific directories. The %Mon hash is supplied so that a recipe has full access to all the monoliths launched for this instrument.

```
orac_process_frame( CurrentRecipe => $STATUS_TEXT,
                    PrimitiveList => $PRIMITIVE_LIST,
                    Frame => $Frm,
                    Group => $Grp,
                    Calibration => $Cal,
                    Engines =>\%Mon,
                    Display => $Display,
                    Beep => $opt_beep,
                    Debug => $opt_debug,
                    CmdLineRecipe => $Override_Recipe,
                    Instrument => $instrument,
                    Batch => 0,
                    RecSuffix => "A,B,C",
                    RecPars => $parameterfile,
                    RecOpts => {},
                   );
```

Additional parameters are provided to configure the recipe environment. Defaults are provided for Debug and Batch. (both false). Those options relate to the `-debug` and `-batch` command line options.

Returns the recipe exit status or throws an exception.

orac_print_config_with_defaults

Wrapper for the `orac_print_configuration` function, but including code to configure default logging switches before configuring the print system.

```
my ($orac_prt, $msg_prt, $err_prt, $ORAC_MESSAGE,
    $PRIMITIVE_LIST, $CURRENT_PRIMITIVE) =
      orac_print_config_with_defaults( \$CURRENT_RECIPE,
                                       \@ARGV, %cloptions );
```

@ARGV contains the command line arguments for the log file. %cloptions are the command line switches. `-debug`, `-showcurrent` and `-log` are used by this routine. `-log` will be read and modified to provide default behaviour.

orac_launch_tk

Attempt to load Tk and create a main window indexed by the identifying string.

```
  $w = orac_launch_tk($win_str, \%opt);
```

This routine can safely be called multiple times.

Returns the top level MainWindow object.

orac_declare_location

Write a file indicating where the pipeline is going to be writing any output data (ie ORAC_DATA_OUT). This file will be written into a directory obtained from the ORAC_LOCATION_DIR or else fall back to the default JAC location of "/jac_sw/oracdr-locations".

The file will be named for the ORAC_INSTRUMENT environment variable and any recipe suffices that are in use. For example "scuba2_450-ql". The file will contain one line with the value of $ORAC_DATA_OUT.

A file is only written if the UT date being used is the current UT date.

```
  orac_declare_location( %options );
```

where %options is the command line options hash.

orac_retrieve_location

Attempt to retrive the pipeline output directory from a file in ORAC_LOCATION_DIR. See orac_declare_location for the details of this file.

orac_print_configuration

This routine setups the orac print system, it takes the $opt_debug and $log_options and the $MW variable and determines which file handles to return

```
  my($orac_prt, $msg_prt, $msgerr_prt, $ORAC_MESSAGE,
     $PRIMITIVE_LIST, $CURRENT_PRIMITIVE)
     = orac_print_configuration(
                               $log_options, $win_str, \$CURRENT_RECIPE
                               \@ORAC_ARGS, %options
                              );
```

The ORAC_ARGS are assumed to be the command line options. `%options` is the options hash. `-debug` and `-showcurrent` are used by this routine.

The tied file handles $orac_prt, $msg_prt and $msgerr_prt are returned, along with the Tk packed variable $ORAC_MESSAGE and a reference to arrays containing the primitive information.

orac_message_launch

This routine creates a message launch system object and configures it, we pass $opt_nomsgtmp and $opt_verbose to the routine to configure the object.

```
  orac_message_launch( $opt_nomsgtmp, $opt_verbose );
```

The message system itself will be initialised when it is required rather than at the start. If we know there is one messsys and we know that it will always be the same one then we can configure it here explicitly. The main reason for doing that is to make sure that it works before starting recipe processing.

orac_start_algorithm_engines

This routine pre-launches the relevant algorithm engines which are always required by the instrument

```
   my ( $Mon )  = orac_start_algorithm_engines( $opt_noeng, $InstObj );
```

it returns a reference to the algorithm engine hash, $Mon.

orac_start_display

This routine is a wrapper for the orac_setup_display() subroutine in ORAC::Basic. It starts the ORAC display unless $nodisplay is set.

```
   my $Display = orac_start_display( $nodisplay, %opt );
```

the routine returns the display object $Display.

Note that an object is returned in all cases, but if display is disabled the display is created in monitor mode.

Additional options are passed to orac_setup_display.

orac_calib_override

This routine creates a calibration object of the specified class and overrides methods as specified in the `--calib` option string.

```
   my $Cal = orac_calib_override( $calclass, @opt_calib, );
```

Multiple calibrations specifications can be supplied. The calibrations are specified as comma separated keyword=value strings or as hash references.

orac_parse_files

This routine parses the text file which has a list of the files to be processed, this should have one filename per line, filenames are assumed to be relative to ORAC_DATA_IN. If there are any duplicates an error will be thrown.

```
   my @obs = orac_parse_files( $opt_files );
```

it returns an array of files to be read.

"#" is a comment character.

orac_parse_recparams

Parse the command line argument specifying recipe parameters and return either a ORAC::Recipe::Parameters object or undef.

```
$params = orac_parse_recparams( $params );
```

orac_process_argument_list

This routine checks that your data exists and decides which data loop approach to use.

```
my $loop =
    orac_process_argument_list( $frameclass, \@obs, %opt );
```

it returns the looping scheme and a list of observations if one does not already exist.

This routine is fairly complex since there are many combinations of `-from`, `-to`, `-skip`, `-loop` and `-list` that interact with each other.

The options hash may contain the following keys: from, to, skip, list and loop. All these are optional as the values may or may not be defined or supplied by the user.

orac_main_data_loop

This routine handles the main data processing.

```
 orac_main_data_loop( \%options, $loop, $instrument, \@obs,
                      $Display, $orac_prt,
                      $ORAC_MESSAGE, $CURRENT_RECIPE, \@PRIMITIVE_LIST,
                      $CURRENT_PRIMITIVE, $Override_Recipe );
```

There are two approaches to the data processing.

(1) The default processing method where data are read in and processed as it arrives and Groups are extended as needed. This has the advantage that the data is processed as it is taken, has very good feedback to the user in real time. The down side is that Groups are processed as soon as possible and in an off-line batch processing envrionment this is very wasteful (why work out the flatfield every time a frame arrives when you simply want to work out the flatfield from the entire group).

(2) The "batch" method where the data are analysed in two passes. First the groups are setup, secondly the frames are processed in each group in turn. This has the advantage that frames can be coadded into a group only once and is the most efficient way of processing data off-line. Note that this presupposes that the primitives are written in such a way that they can spot the last member of the group (via the lastmember method). Grp Primitives without this check will probably fail since the some of the members will not have been processed even though the group contains many members.

One other issue is calibration -- in principal all calibration groups should be processed before observation groups and currently this is not supported (only important when calibrations are taken after the observation).

Batch mode can be summarised as

```
- Read in all frames and allocate groups
- Loop over all groups Loop over all frames in
  group process frames
```

Default mode is

```
- Loop over all frames
- Allocate groups
- process frames
```

Batch mode can be turned on with the -batch switch.

Returns a hash containing information on the error status from all the frames that were processed. The hash can be analyzed using orac_print_recipe_summary.

orac_store_recipe_status

Translates a return status from "orac_process_frame" into a hash entry for tracking statistics, and recording the filenames where the processing failed or the supplied data were deemed bad.

```
orac_store_recipe_status( \%ongoing, $status, $filename );
```

orac_print_recipe_summary

Print out the recipe summary using the hash generated by orac_store_recipe_status.

```
$exstat = orac_print_recipe_summary( $color, \%Stats );
```

Returns 0 if all recipes processed successfully and non-zero if some failed. This value can be passed directly to exit().

Returns 0 if the stats hash is empty and -1 if no hash reference is supplied.

COPYRIGHT

D.8 ORAC::Display::Base

Base class for ORAC display interface

SYNOPSIS

```
  use ORAC::Display::Base;
```

DESCRIPTION

Provides the generic methods for handling ORAC Display devices. The generic routines (those worth inheriting) deal with display device name allocation (eg mapping a device number to a real device).

PUBLIC METHODS

Constructor

new

Base class constructor. Can be called as SUPER::new() from sub-classes. Accepts a configuration hash as input in order to initialise extra instance data components of the class that are required by sub-classes.

```
  $a = new ORAC::Display(a => 'b', c => 'd');
```

This constructor does not attempt to launch a display device. That is up to the sub-classes.

Accessor Methods

dev

Method for handling the hash of device name mapping. ie Which device name (as required for each Display interface, eg '.rtd0', 'xwindows;$$') is associated with the ORAC name (eg '0','1','default').

The hash reference is returned when called with no arguments:

```
  $href = $self->dev;
```

The value associated with the supplied key is returned if one argument is provided:

```
  $value = $self->dev('key');
```

The supplied value is stored in key if two arguments are supplied:

```
  $self->dev('key','value');
```

Undefined values are accepted.

General Methods

window_dev

Returns the device id (eg GWM device name or RTD window name) associated with window 'win'. If 'win' is undefined a new window is launched, the id stored in the hash and the new id returned. (see the launch_dev() method). If this is the first time the routine is called (ie the only window name present is 'default', the name of the default window is associated with window win.). We go through this hoop so that devices will open a window before the user has associated their user-defined name with the actual window name.

```
  $name = $self->window_dev('win');
```

If the windows were launched with bad status we should set the device name to something recognisable as bad since status is not returned.

SEE ALSO

ORAC::Display::GAIA, *ORAC::Display::KAPVIEW*

COPYRIGHT

Copyright (C) 1998-2000 Particle Physics and Astronomy Research Council. All Rights Reserved.

D.9 ORAC::Display::GAIA

ORAC interface to GAIA

SYNOPSIS

```
  $disp = new ORAC::Display::GAIA;

  $disp->image($file);
```

DESCRIPTION

ORAC interface to the the GAIA (ESO Skycat) display tool. Provides methods for displaying images.

Available options are:

IMAGE - display image in GAIA window

PUBLIC METHODS

Constructor

new

Object constructor. The constructor starts up a new version of GAIA (if one is not running) and connects via a socket.

The program aborts if there is an error launching or contacting gaia.

Accessor Methods

launchable

Whether or not GAIA can be automatically launched.

```
$gaia->launchable( 0 );
```

Defaults to true. If this is set to false (0), then a new GAIA will never be started.

sock

Returns or sets the socket to Gaia. Private to this class.

```
$sock = $gaia->sock();
```

This is usually IO::Socket object. This socket is automatically added to the IO::Select object returned by the `sel` method. (and all previous sockets registered with the IO::Select object are removed).

sel

Returns the IO::Select object associated associated with the current socket.

```
$select = $gaia->sel();
```

This object is used to determine whether the GAIA process can be contacted through the established socket connection.

use_remote_gaia

Controls whether we are allowed to connect to a GAIA process that is already running on a remote machine. By default this is allowed (true) but in some cases you may not want to connect to a remote GAIA. For example, at UKIRT, the display must be sent to the machine running the pipeline and not one of the other GAIAs that are running on separate machines for QuickLook and general data inspection.

General Methods

create_dev

Clone a new GAIA window and associate it with 'win'. This is different to launching a new display device (ie running up GAIA itself).

```
$status = $Display->create_dev($win, $name);
```

For GAIA (V <= 2.3-2) the device name ($name) must be an integer. (enforced if the newdev() method is used).

ORAC status is returned.

launch

Connect to a pre-existing Gaia process or launch a new Gaia process. If the first connection attempt fails, launches a new gaia process. After this, attempts to connect to a new gaia process every 3 seconds and attempts to launch a new gaia process every 60 seconds. A maximum of 5 attempts are made (5 minutes) to launch a new Gaia process before giving up.

There is no return status -- the program croaks if it can not get a connection to GAIA !!

Whilst it is waiting, does not attempt to keep a `Tk` event loop running.

configure

Load the startup image into GAIA. Essentially used to test that GAIA can display images correctly.

Returns ORAC status.

send_to_gaia

Sends the supplied command to gaia. Any response from Gaia is returned.

```
($status, $return_string) = $obj->send_to_gaia('command');
($status, @return_strings) = $obj->send_to_gaia(@commands);
```

The returned status is translated into an ORAC status (either ORAC__OK or ORAC__ERROR). On error, the return_string contains the error message. The status returned is the status of the last command processed by GAIA.

newdev

Returns the name to be used for the new GAIA window based on the supplied window name.

```
 $name = $obj->newdev($win);
```

Currently, for gaia, the argument is ignored. The name is simply returned as an integer calculated from the number of devices already stored in the object.

DISPLAY METHODS

image

Routine to display images in Gaia. Note that the full file name is required. If an image name does not include an extension then '.sdf' is appended. (ie NDF is assumed).

Takes a file name and arguments stored in a hash.

```
$disp->image("filename", \%options)
$disp->image("filename", { WINDOW => 2 });
```

Currently no image sectioning is supported. Display range can be adjusted with ZAUTOSCALE, ZMIN and ZMAX.

Component can be selected with COMP option. DATA is the default. QUALITY and VARIANCE are supported.

Note that for GAIA, ZAUTOSCALE implies a 95 percent cut level and not 100 percent.

Will attempt to relaunch GAIA if it can not be contacted. Will attempt to create a new clone window if a clone can not be contacted even though it has been used previously.

ORAC status is returned.

SEE ALSO

ORAC::Display::Base, *ORAC::Display::KAPVIEW*, *ORAC::Display*, *IO::Socket*

COPYRIGHT

D.10 ORAC::Display::KAPVIEW

ORAC-DR interface to Kapview (KAPPA)

SYNOPSIS

```
use ORAC::Display::KAPVIEW;
$disp = new ORAC::Display::KAPVIEW;

$disp->image($file, { XAUTOSCALE => 1});
```

DESCRIPTION

ORAC interface to KAPPA Kapview. Provides methods for displaying images and spectrum with Kapview.

Available options are:

```
IMAGE - display image using DISPLAY
GRAPH - display graph using LINPLOT
SIGMA - display scatter plot with a Y-range of +/- N sigma.
DATAMODEL - Display data (as points) with a model overlaid
HISTOGRAM - Histogram of values in data array
VECTOR - Display image + vectors
```

PUBLIC METHODS

Constructor

new

Object constructor. The constructor starts up a new version of kapview, starts a GWM window and displays the startup logo.

The program aborts if there is an error launching kapview.

The message system must be running so that Kapview can be configured. (AMS is started if needed.)

Accessor Methods

engine_launch_object

Returns the `ORAC::Msg::EngineLaunch` object that can be used to launch algorithm engines as required by the particular conversion.

```
 $messys = $self->messys_launch_object;
```

kappa

Messaging object associated with the kappa_mon monolith. This is used by some of the modes in order to determine display related values (e.g. statistics to determine plotting ranges for SIGMA, dimension compression with COMPAVE).

A KAPPA messaging object is created if the object is undefined.

Note also that the HISTOGRAM task is present in the KAPPA monolith rather than in the KAPVIEW monolith.

ndfpack

Messaging object associated with the ndfpack_mon monolith. This is used by some of the modes in order to reshape date arrays (e.g. in SIGMA mode - reshape is run to convert to 1-d).

A NdfPack messaging object is created if the object is undefined.

polpack

Messaging object associated with the polpack_mon monolith. This is used by the VECTOR mode to plot vectors from catalogues.

Note that this is technically not part of the KAPVIEW system. It is here for convenience since in all cases POLPLOT is better than VECPLOT for vector plotting.

A Polpack messaging object is created if the object is undefined.

Returns undef if polpack_mon is not available.

regions

A hash containing the mapping of region name (number) to AGI picture label.

Returns hash reference in scalar context, full hash in array context. Contents can be modified by directly using the hash reference (in order to modify specific entries) or completely rewritten by supplying a hash as argument.

```
    $hashref = $self->regions;
    %hash = $self->regions;
    $self->regions(%hash);
    $self->regions->{Key} = "value";
```

lookup_table

Set or retrieve the previous lookup table.

obj

Messaging object associated with the Kapview display object.

General Methods

newdev

Given 'win', calculates a new device name that should be unique for each 'win'.

```
   $dev = $Display->newdev($win);
```

calc_centre_region

create_dev

Start the GWM window associated with the supplied window. In general this is used by the startup configuration. The lookup table is configured by this routine (so that this routine really does start the GWM window).

Currently the GWM window itself is not started directly by this routine (since KAPVIEW will automatically open the specified device if one is not running).

The only reason to use this routine to actually START a window is that it will give us some control over the colour allocation and allow us to set the window name.

ORAC status is returned.

```
    $status = $Display->create_dev($win);
```

Currently, the method dies if the device can not be successfully created.

launch

This method starts the kapview monolith and stores the associated Task object.

configure

Load a startup image. This tests the system to make sure that images can be displayed and that the colour map is loaded.

Returns ORAC status.

config_regions

This method configures the display regions so that they can be selected later by select_region.

```
$status = $self->config_regions($window);
```

A window name must be supplied.

The regions are defined as follows:

```
0 - full screen
1 - top left
2 - top right
3 - bottom left
4 - bottom right
5 - left
6 - right
7 - top
8 - bottom
17:32 - position in 4x4 grid (starting top left)
```

The picture labels are stored in the regions() array.

select_region

Selects the requested region as the current region in the display system by using a supplied hash.

```
$device = $Display->select_region(%options);
```

Returns undef without action if the REGION keyword is not available (since have no idea where to put it) or if REGION is not in the allowed range. Otherwise the name of the device containing the selected region is returned. undef is returned if no arguments are supplied.

If the window name is not supplied (WINDOW) then 'default' is assumed.

select_section

This method converts a file name and options hash into a filename with an attached NDF section.

```
$newfile = $Display->select_section($file, \%options, $dimensionality);
```

An optional 3rd argument can be used to specify the required dimensionality. If the number of dimensions in the data file is greater than that requested, sections in higher dimensions are set to 1 by compressing the undesired dimension (with the assumption that KAPPA will discard axes with 1 pixel). The desired dimension is specified with the CUT option. For example, a graph can be displayed from a 2-D image by displaying a cut in the X direction (averaging over the Ys).

If the number of dimensions in the data file is fewer than that requested, a warning message is printed but we continue in the hope that KAPPA will work something out....

The return value is the original filename with the NDF section attached.

Relevant keywords in options hash:

```
CUT  - Specify the significant dimension[s] (X,Y,3,4,5)
       Should be a comma-separated list specifying
       dimensionality - number of entries should equal the
       requested dimensionality. For a graph only 1 value
       is required since a graph is 1-D
XMIN/XMAX - X pixel max and min values
YMIN/YMAX - Y pixel max and min values
XAUTOSCALE - Use autoscaling for X?
YAUTOSCALE - Use autoscaling for Y?
```

If Xautoscale and Yautoscale are true, no section command is appended. If the XAUTOSCALE/YAUTOSCALE/nAUTOSCALE keywords can not be found they are assumed to be true. If CUT is not specified the first slice is selected (e.g. a NDF section of N,1,1,1).

For data arrays with N>2, the leading letter is dropped and replaced by the dimension number. For example,

```
3MIN/3MAX - pixel range of the 3rd dimension
4AUTOSCALE - autoscale the 4th dimension?
```

For NDFs the maximum dimensionality is 7.

The bounds of the input file are compared to the supplied bounds. If any of the requested bounds are exceeded, the maximum value will be used instead.

Returns undef on error. The unmodified file name is returned if no options hash can be found.

Returns the following:

```
No CUT requested + auto-scaling
  returns the original filename.
No CUT requested + some dimension ranges specified
  returns the original filename with an NDF section.
  Dimensions above the requested dimensionality are set to the
  min value in the section (1 if not specified)
CUT requested but dimensionality of data matches requested
  dimensionality.
  Just return the file + any relevant section
CUT + auto-scaling + image too large
  Data file is collapsed down to required size keeping the specified
  dimensions and averaging over the rest. A new temporary filename
  is returned.
CUT + some ranges specified + image too large
  NDF section constructed and then the data file is collapsed
  down to the required size. A new temporary file is generated.
CUT + range + image + one pixel selected
  If the non-cut dimensions have min=max a section is
  sufficient and no averaging required.
```

The temporary files themselves are added to a global class array and removed by the destructor.

Note that this routine does not remove the temporary filename. This is probably a bug. Should probably create some kind of object that will have a destructor that removes the file rather than using a simple file name. (an ORAC::TempFile)?

set_lut

Set the given LUT.

```
$status = $Display->set_lut($device, $lut);
```

This method also calls lookup_table to store the name of the specified LUT.

DISPLAY METHODS

image

Display an image. Takes a file name and arguments stored in a hash. Note that currently it does not take a format argument and NDF is assumed.

Recognised options:

```
XMIN/XMAX - X pixel max and min values
YMIN/YMAX - Y pixel max and min values
XAUTOSCALE - Use autoscaling for X?
YAUTOSCALE - Use autoscaling for Y?
ZMIN/ZMAX  - Z-range of greyscale (data units)
ZAUTOSCALE - Autoscale Z?
KEY        - Display key to colour table?
COMP       - Component to display (Data (default), Variance or Error)
LUT        - Color lookup table name [default: 'bgyrw']
BADCOL     - Bad color name [default: 'grey50']
```

Default is to autoscale.

ORAC Status is returned.

graph

Display a 1-D plot.

If the data are not 1-D, a section is taken that assures 1-D (e.g. NDF section= :,1,1,1 for 4D data).

Takes a file name and arguments stored in a hash. Note that currently it does not take a format argument and NDF is assumed.

Display keywords:

```
XMIN/XMAX  - X-pixel range of graph
XAUTOSCALE - Autoscale pixel range?
YMIN/YMAX  - Y-pixel range of graph (in pixels)
```

```
YAUTOSCALE - Autoscale Y-axis
YMIN/YMAX  - Z-range of graph (in data units)
YAUTOSCALE - Autoscale Z-axis
CUT        - Decide which direction is the primary axis
             Can be X,Y,3,4,5 (for higher-dimensional data sets)
             For a 1-D data set (or section), this value is ignored
COMP       - Component to display (Data (default), Variance or Error)
ERRBAR     - Plot error bars or not (if variance information is
             present)
YLOG       - Plot Y-axis on a logarithmic scale
MULTILINE  - Plot multiple lines (using mlinplot)
ABSAXIS    - Axis selection (when using MULTILINE)
HLRANGE    - Range (NDF section axis specifier) to highlight
```

Default is to autoscale. Note that the X/Y cuts are converted to a 1-D slice before displaying by averaging over the section.

For example:

```
XMIN=5 XMAX=5 YAUTOSCALE=YES
```

would display column 5 (i.e. the whole of Y for X=5). [CUT is irrelevant since the resulting image section is 1-D], and

```
XAUTOSCALE=YES YMIN=20 YMAX=30 CUT=X
```

would display the average of rows 20 to 30 for each X.

Need to add way of controlling line style (e.g. replace with symbols)

ORAC status is returned.

chanmap

Displays a channel map of a central region of a cube.

Recognized options:

```
XMIN/XMAX  - X pixel min and max values.
YMIN/YMAX  - Y pixel min and max values.
ZMIN/ZMAX  - Z-range of greyscale (data units)
XAUTOSCALE - Use autoscaling for X?
YAUTOSCALE - Use autoscaling for Y?
ZAUTOSCALE - Use autoscaling for Z?
WIDTH      - Width of central region to use, in percent.
AXIS       - Axis to collapse over.
NCHAN      - Total number of channels to display.
SHAPE      - Number of channels to display along X-axis.
```

WIDTH defaults to 100. AXIS defaults to 3. NCHAN defaults to 9. SHAPE defaults to 3.

ORAC status is returned.

cubecentre

Collapse the central region of a cube along one axis, and display an image of the collapse region.

Recognised options:

```
XMIN/XMAX  - X pixel min and max values.
YMIN/YMAX  - Y pixel min and max values.
ZMIN/ZMAX  - Z-range of greyscale (data units)
XAUTOSCALE - Use autoscaling for X?
YAUTOSCALE - Use autoscaling for Y?
ZAUTOSCALE - Use autoscaling for Z?
WIDTH      - Width of central region to use, in percent.
AXIS       - Axis to collapse over.
```

WIDTH defaults to 100. AXIS defaults to 3.

ORAC status is returned.

contour

Display contours of a 2-D data set.

Recognised options:

```
XMIN/XMAX - X pixel max and min values
YMIN/YMAX - Y pixel max and min values
XAUTOSCALE - Use autoscaling for X?
YAUTOSCALE - Use autoscaling for Y?
ZMIN/ZMAX  - Z-range of greyscale (data units)
ZAUTOSCALE - Autoscale Z?
NCONT      - Number of contours
COMP       - Component to display (Data (default), Variance or Error)
KEY        - Display key to contour levels?
```

Default is to autoscale.

ORAC status is returned.

sigma

Display a scatter plot of the data with Y range of N-sigma (sigma is derived from the data) with dashed lines overlaid at the X-sigma points.

By default a range of +/-5 sigma with dashed lines at +/-3 sigma are used.

These values can be overriden by using the RANGE and DASHED keywords.

Takes a file name and arguments stored in a hash. Note that currently it does not take a format argument and NDF is assumed.

If we are running KAPPA 0.13, the NDF is converted to 1-DIM with the KAPPA/RESHAPE command before displaying.

ORAC status is returned.

datamodel

Display mode where the supplied filename is plotted as individual points and a model is overlaid as a solid line. This can be used to determine the goodness of fit of data and model.

The model filename is derived from the input filename (a _model extension is expected). The data are displayed if the model file can not be found.

Takes a file name and arguments stored in a hash. Note that currently it does not take a format argument and NDF is assumed.

Option keywords:

```
XMIN/XMAX  - X-pixel range of graph
XAUTOSCALE - Autoscale pixel range?
ZMIN/ZMAX  - Y-range of graph (in data units)
ZAUTOSCALE - Autoscale Y-axis
COMP       - Component to display (Data (default), Variance or Error)
```

Default is to autoscale on the data (the model may not be visible).

If the input file is greater than 1-D, the section is automatically converted to 1-D by selecting the ?MIN slice from each of the higher axes (e.g. the value specified in YMIN, 3min...).

ORAC status is returned.

histogram

Display a histogram of the data values present in the data array.

Takes a file name and arguments stored in a hash. Note that currently it does not take a format argument and NDF is assumed.

Arguments:

```
XMIN/MAX   - minimum/maximum x-pixel value
XAUTOSCALE - Use full X-range
YMIN/YMAX  - minimum/maximum x-pixel value
YAUTOSCALE - use full Y-range
ZMIN/ZMAX  - Z range of histogram (data units)
ZAUTOSCALE - use full Z-range
NBINS      - Number of bins to be used for histogram calculation
COMP       - Component to display (Data (default), Variance or Error)
SIGMA      - Number of sigma to clip data before computing histogram
TITLE      - title string to label the histogram
```

Default is for autoscaling and for NBINS=20. Note that the presence of SIGMA overrides ZMIN/ZMAX.

ORAC status is returned.

vector

Vectors are overlaid on an image. The supplied file is displayed and vectors are then drawn. The vector information is expected to be stored in the ORAC extension of the supplied file (in .P and .THETA NDFs) or, preferably, in a catalogue of the same name as the I image. POLPLOT is used for display if the catalogue is available.

Recognised options:

```
XMIN/XMAX  - X pixel max and min values
YMIN/YMAX  - Y pixel max and min values
XAUTOSCALE - Use autoscaling for X?
YAUTOSCALE - Use autoscaling for Y?
ZMIN/ZMAX  - Z-range of greyscale (data units)
ZAUTOSCALE - Autoscale Z?
ANGROT     - angle to add to all vectors
MULTIVECTOR- Plot multi-coloured vectors (yellow with blue trim)?
```

Default is to autoscale.

ORAC status is returned.

SEE ALSO

ORAC::Display, *ORAC::Display::GAIA*

COPYRIGHT

D.11 ORAC::Error

Exception handling in an object orientated manner.

SYNOPSIS

```
use ORAC::Error qw /:try/;
use ORAC::Constants qw /:status/;

# throw an error to be caught
throw ORAC::Error::UserAbort( $message, ORAC__ABORT );
throw ORAC::Error::FatalError( $message, ORAC__FATAL );

# record and then retrieve an error
do_stuff();
my $Error = ORAC::Error->prior;
ORAC::Error->flush if defined $Error;
```

```
sub do_stuff {
    record ORAC::Error::FatalError( $message, ORAC__FATAL);
}


# try and catch blocks
try {
   stuff();
}
catch ORAC::Error::UserAbort with
{
    # normally we just want to catch and then ignore UserAborts
    my $Error = shift;
    orac_exit_normally();
}
catch ORAC::Error::FatalError with
{
    # its a fatal error
    my $Error = shift;
    orac_exit_normally($Error);
}
otherwise
{
   # this block catches croaks and other dies
   my $Error = shift;
   orac_exit_normally($Error);
}; # Don't forget the trailing semi-colon to close the catch block
```

DESCRIPTION

`ORAC::Error` is based on a modifed version of Graham Barr's `Error` package, and more documentation about the (many) features present in the module but currently unused by ORAC-DR can be found in the documentation for that module.

As with the `Error` package, `ORAC::Error` provides two interfaces. Firstly it provides a procedural interface to exception handling, and secondly `ORAC::Error` is a base class for exceptions that can either be thrown, for subsequent catch, or can simply be recorded.

If you wish to throw an `FatalError` or `UserAbort` then you should also `use ORAC::Constants qw / :status /` so that the ORAC constants are available.

PROCEDURAL INTERFACE

`ORAC::Error` exports subroutines to perform exception handling. These will be exported if the `:try` tag is used in the `use` line.

try BLOCK CLAUSES

`try` is the main subroutine called by the user. All other subroutines exported are clauses to the try subroutine.

The BLOCK will be evaluated and, if no error is throw, try will return the result of the block.

`CLAUSES` are the subroutines below, which describe what to do in the event of an error being thrown within BLOCK.

catch CLASS with BLOCK

This clauses will cause all errors that satisfy `$err->isa(CLASS)` to be caught and handled by evaluating `BLOCK`.

`BLOCK` will be passed two arguments. The first will be the error being thrown. The second is a reference to a scalar variable. If this variable is set by the catch block then, on return from the catch block, try will continue processing as if the catch block was never found.

To propagate the error the catch block may call `$err->throw`

If the scalar reference by the second argument is not set, and the error is not thrown. Then the current try block will return with the result from the catch block.

otherwise BLOCK

Catch *any* error by executing the code in `BLOCK`

When evaluated `BLOCK` will be passed one argument, which will be the error being processed.

Only one otherwise block may be specified per try block

CLASS INTERFACE

CONSTRUCTORS The `ORAC::Error` object is implemented as a HASH. This HASH is initialized with the arguments that are passed to it's constructor. The elements that are used by, or are retrievable by the `ORAC::Error` class are listed below, other classes may add to these.

```
-file
-line
-text
-value
```

If `-file` or `-line` are not specified in the constructor arguments then these will be initialized with the file name and line number where the constructor was called from.

The `ORAC::Error` package remembers the last error created, and also the last error associated with a package.

throw ([ARGS])

Create a new `ORAC::Error` object and throw an error, which will be caught by a surrounding `try` block, if there is one. Otherwise it will cause the program to exit.

`throw` may also be called on an existing error to re-throw it.

with ([ARGS])

Create a new `ORAC::Error` object and returns it. This is defined for syntactic sugar, eg

```
die with ORAC::Error::FatalError ( $message, ORAC__FATAL );
```

record ([ARGS])

Create a new `ORAC::Error` object and returns it. This is defined for syntactic sugar, eg

```
record ORAC::Error::UserAbort ( $message, ORAC__ABORT )
    and return;
```

METHODS

prior ([PACKAGE])

Return the last error created, or the last error associated with `PACKAGE`

```
my $Error = ORAC::Error->prior;
```

flush ([PACKAGE])

Flush the last error created, or the last error associated with `PACKAGE`.It is necessary to clear the error stack before exiting the package or uncaught errors generated using `record` will be reported.

```
$Error->flush;
```

OVERLOAD METHODS

stringify

A method that converts the object into a string. By default it returns the `-text` argument that was passed to the constructor, appending the line and file where the exception was generated.

value

A method that will return a value that can be associated with the error. By default this method returns the `-value` argument that was passed to the constructor.

PRE-DEFINED ERROR CLASSES

ORAC::Error::FatalError

Used for fatal errors where we want the pipeline to die with cause. This class can be used to hold simple error strings and values. It's constructor takes two arguments. The first is a text value, the second is a numeric value, `ORAC__FATAL`. These values are what will be returned by the overload methods.

ORAC::Error::UserAbort

Used for user generated pipeline aborts, which are handled slightly differently than fatal errors generated by the pipeline itself. The constructor for a `UserAbort` is similar to that for a `FatalError` except that the numeric value `ORAC__ABORT` is passed.

ORAC::Error::TermProcessing

Terminate processing of this recipe but continue on as if everything is fine. This can be used to stop a recipe early but will not trigger any long term errors to propagate through the pipeline.

ORAC::Error::TermProcessingErr

As for TermProcessing except that whilst the pipeline will continue to run the exit status of the pipeline will be non-zero to indicate that a problem was found. This can be used when the pipeline can work around the problem but would like the final exit status to reflect that there is a problem to be investigated.

KNOWN PROBLEMS

`ORAC::Error` which are thrown and not caught inside a `try` block will in turn be caught by `Tk::Error` if used inside a Tk environment, as will `croak` and `die`. However if is a `croak` or `die` is generated inside a try block and no `otherwise` block exists to catch the exception it will be silently ignored until the application exits, when it will be reported.

ACKNOWLEDGMENTS

This class is a slightly modified, with the addition of the `flush` method, version of Graham Barr's (gbarr@pobox.com) `Error` class. That code was in turn based on code written by Peter Seibel (peter@weblogic.com) and Jesse Glick (jglick@sig.bsh.com).

D.12 ORAC::Inst::Defn

Definition of instrument class dependencies

SYNOPSIS

```
use ORAC::Inst::Defn;

@pars = orac_determine_inst_classes( $instrument );
orac_determine_initial_algorithm_engines
orac_determine_recipe_search_path
orac_determine_primitive_search_path
orac_engine_description
orac_messys_description
orac_configure_for_instrument( $instrument, \%options );
```

DESCRIPTION

This module provides all the instrument specific initialisation information. This is the information required by ORAC-DR in order to configure itself before the data detection loop can begin.

This module provides information on class hierarchies, recipe search paths and initialisation or algorithm engines.

All instrument dependencies are specified in this module.

FUNCTIONS

The following functions are provided:

orac_determine_inst_classes

Given an ORAC instrument name, returns the class names to be used for frames, groups, calibration messaging. The classes are used so that objects can be instantiated immediately.

```
($frameclass, $groupclass, $calclass, $instclass) =
      orac_determine_inst_classes( $instrument );
```

The function dies if the classes can not be used. An empty list is returned if the instrument is not known to the system.

Returns the Frame class in scalar context.

orac_list_generic_observing_modes

Returns a list of the standard observing modes supported by ORAC-DR. Currently the list includes just "imaging" and "spectroscopy" (implicitly assumed to be near-infrared).

Specific instruments always have their own modes.

The list is determined by looking in the recipe directory for directories that contain all lower case characters. Upper case characters imply a specific instrument rather than a generic mode. This is probably over-the-top given that the search path functions (below) have to assume that they know the answer.

orac_determine_recipe_search_path

Returns a list of directories that should be searched in order to locate recipes for the specified instrument.

```
@paths = orac_determine_recipe_search_path( $instrument );
```

Root location is specified by the `ORAC_DIR` environment variable.

Any instruments that start with "PICARD" will only include the PICARD search paths.

orac_determine_primitive_search_path

Returns a list of directories that should be searched in order to locate primitives for the specified instrument.

```
@paths = orac_determine_primitive_search_path( $instrument );
```

Root location is specified by the `ORAC_DIR` environment variable.

orac_determine_calibration_search_path

Returns a list of directories that should be searched in order to locate calibration files for the specified instrument.

```
@paths = orac_determine_calibration_search_path( $instrument );
```

Root location is specified by the `ORAC_CAL_ROOT` environment variable. ORAC_DATA_CAL is included if it is set explicitly.

orac_determine_initial_algorithm_engines

For the supplied instrument name, returns a list containing the names of engines to be launched by the pipeline prior to executing any recipes. This is used so that engines that are always used will be available at the start of execution rather than being launched on demand. This approach provides a slight efficiency gain over starting each engine on demand.

```
@engines = orac_determine_initial_algorithm_engines( $instrument)
```

In principal this list can be empty if no pre-launching is required.

orac_configure_for_instrument

This routines configures the user environment (e.g. %ENV) for the instrument, it is called by Xoracdr to replace functionality present in the c-shell setup scripts.

```
 orac_configure_for_instrument( $instrument, \%options );
```

orac_guess_instrument

Given a Frame object (assumed to be a generic type frame with a translated FITS header) make a guess at the corresponding ORAC_INSTRUMENT that should be used.

```
$guess = orac_guess_instrument( $Frm );
```

Useful for converting a base class or a PICARD variant to a specific type.

Returns undef if none can be guessed.

orac_engine_description

Returns the details for a specified algorithm engine.

```
%details = orac_engine_description("polpack_mon");
```

The hash that is returned contains information on the class to be used to launch the engine and the location of the engine. In future it may also return the messaging system required for the engine to function. The hash currently has the following keys

CLASS

The name of the class to be used for this engine. (e.g. `ORAC::Msg::Task::ADAM`).

PATH

The location of the engine in the file system. If this is a code reference it should be executed immediately prior to launching the monolith to configure associated parameters correctly and to return the actual path. Additionally, if the helper task is executed it returns a reference to a cleanup subroutine. See §D.12.

MESSYS
The name of the message system required to contact the engine. See `orac_messys_description` below for details.

Returns an empty list on error.

orac_messys_description

Returns the details for a specified message system.

```
  %details = orac_messys_description("AMS");
```

The hash that is returned contains information on the class to be used to initialise the message system. It has the following keys

CLASS
The name of the class to be used for this message system (e.g. `ORAC::Msg::ADAM::Control`).

Returns an empty list on error.

HELPER TASKS

Some algorithm engines need to be configured in a slighlty more complex way than providing a simple path to the engine. This section describes specific functions that return the name of the path whilst also configuring the program before launch. For example, can be used to create a temporary directory for special output. The helper tasks accept no arguments and are required to return a path to an engine and a reference to a subroutine to be exected when the object has been launched. This allows for cleanup code to be executed and are usually closures.

p4_helper

Helper task for the CGS4-DR P4 display system.

```
  ($path, $cleanup) = p4_helper;
```

COPYRIGHT

D.13 ORAC::Msg::EngineLaunch

Launch engines on demand

SYNOPSIS

```
use ORAC::Msg::EngineLaunch;

$eng = new ORAC::Msg::EngineLaunch;

$obj = $eng->engine("polpack_mon");
$eng->detach("polpack_mon");
(@ok, @nok) = $eng->contact_all;

tie %Mon, "ORAC::Msg::EngineLaunch";
$obj = $Mon{"polpack_mon"};
delete $Mon{"polpack_mon"};
```

DESCRIPTION

This class provides a means of launching arbitrary algorithm engines on demand. If an engine has not previously been launched the class will start it, if it has been launched it will retrieve the current object. The algorithm engines will be `ORAC::Msg` task objects (eg *ORAC::Msg::ADAM::Task*). This allows engines to be launched only when required to minimize resource demand.

It is also possible to tie the class to a hash. This allows for a non-object oriented approach where the engine can be launched simply by accessing the engine through the hash.

METHODS

The following methods are provided:

Constructor Object constructors.

new

Instantiate a new object ready for launching.

```
$launch = new ORAC::Msg::EngineLaunch( $unique );
```

Since, in general, it is convenient for all parts of the code to have access to previously launched engines, the default behaviour is for the constructor to return the same object reference each time it is called. If it is required for a completely new object to be created each time the argument must be set to true.

ORAC-DR usually requires that access is provided to all previously launched engines for efficiency (and to prevent name clashes).

Accessor Methods

engine

Retrieve the object associated with the specified engine, launching it if required.

```
$obj = $launch->engine("polpack_mon");
```

`undef` is returned if the engine could not be launched.

The engine object can be stored if two arguments are used. A rudimentary check is made to make sure that the object is a reference and that the `contactw` method is supported. It is not possible to check a true ISA relationship. If the object does not satisfy this condition it is not stored and a warning is raised with "-w".

```
$launch->engine("polpack_mon", $object);
```

Returns a hash reference containing all the currently launched engines if called without arguments.

```
$launched = $launch->engine;
```

engine_id

The message system identifier. This is used by some message systems (e.g. ADAM) to indicate a specific identifier that should be used to name the engine in the message system. This allows, for example, the pipeline to attach to an engine that has been launched outside of the pipeline infrastructure. For engines launched by the pipeline each new identifier must be unique for each pipeline and for each repeat monolith launch (in the case where engines die and are restarted).

This method is used to store the previous id for each engine so that a new id can be generated.

```
$id = $launch->engine_id( $engine );
$launch->engine_id( $engine, $id );
```

The `engine_inc` method should be used to generate a new id.

If no arguments are supplied a reference to the hash of IDs is returned. `undef` is returned if an id is requested for an engine that has not been launched.

messys_launch_object

Returns the `ORAC::Msg::MessysLaunch` object that can be used to initialise message systems as required by the particular algorithm engines.

```
$messys = $self->messys_launch_object;
```

General Methods

contact_all

Runs the `contactw` method on each registered engine. Can be used to make sure that all the registered engines are okay. If an engine can not be contacted it is removed from the object.

Returns two arrays, one for engines that could be contacted and one for engines that could not be contacted.

```
  ($okay, $notokay) = $launch->contact_all;
```

In a scalar context simply returns true if all engines could be contacted. Also returns true if there are no registered engines.

```
  $all_okay = $launch->contact_all;
```

The message system timeout is reduced to 30 seconds whilst waiting for contact. It is subsequently reset afterwards. This allows the pipeline configuration of timeout to vary from that required simply to check that the monolith can be contacted.

detach

Disassociate the named engine from the object. This can be used if an engine has crashed and it is necessary to launch a new engine next time.

```
  $launch->detach( $engine );
```

launch

Launch the specified monolith.

```
  $obj = $launch->launch( $engine );
```

The engine object is stored in the class. Returns undef on error.

The routine does not return until the engine has completed loading (i.e. the `contactw` method returns successfully). This is less efficient than launching all the monoliths and then waiting for them but it is the price paid for launching on demand.

This overhead can be overcome by pre-launching engines that are known to be required and launching optional engines on demand. If multiple engine names are supplied to this method they will all be launched at once without waiting for each one in turn. A hash is returned containing all the objects that were launched.

```
  %obj = $launch->launch( $engine1, $engine2 );
```

If multiple engines are launched simultaneously, the status of the engines must then be checked explicitly using the `contact_all` method.

If the engines are launched outside this infrastructure they can be registered with the object using the `engine` method for the object, and `engine_id` method to register the messaging name (if appropriate).

If a request is made to launch an engine that has been launched previously the request returns the current engine object. Use `detach` to force a reload.

engine_inc

Return a new ID for the specified engine.

```
  $id = $self->engine_inc( $engine );
```

The current ID is updated (see `engine_id` for more details).

TIED INTERFACE

This class also provides a means of tieing an object to a standard perl hash allowing for transparent access to engines.

A hash can be tied to an object by using the `tie` function:

```
  tie %Mon, "ORAC::Msg::Engine::Launch";
```

It is also possible to tie a hash to an existing object:

```
  tie %Mon, ref($object), $object;
```

The following can be used to retrieve the object associated with "polpack_mon" launching the engine if necessary:

```
  $object = $Mon{"polpack_mon"};
```

Engines can be dissassociated from the object using the standard hash `delete` command:

```
  delete $Mon{"polpack_mon"};
```

`exists`, `keys` and `each` are supported.

Note that `exists` will *not* launch a monolith. It can only be used to check that one has already been launched.

In addition, it is possible to explicitly set entries in the hash. A rudimentary check is made to check that the stored entry is an object that can invoke a "contactw" method but it is not possible to check that the object is of the correct type (since there is currently no complete inheritance tree for engines). If the argument is not okay the object a warning will be issued under "-w".

```
$Mon{engine} = $some_object;
```

A reference to the hash still has access to the tied hash. A copy of the hash (e.g. `%New = %Old`) will copy the contents of the hash without copying the tie. In order to copy the hash and retain the tie, it is necessary to tie the new hash rather than copying it.

```
$object = tied %Mon;
tie %New, ref($object), $object;
```

COPYRIGHT

D.14 ORAC::Msg::MessysLaunch

Generic interface for initialising message systems

SYNOPSIS

```
use ORAC::Msg::MessysLaunch;

$msl = new ORAC::Msg::EngineLaunch;

$obj = $msl->messys( 'AMS' );
%objs = $msl->messys_active;
```

DESCRIPTION

This class provides a generic interface to the messaging systems supported by ORAC-DR. The knowledge of how to setup and initialise all the supported messaging systems is included in this class.

The message systems are started on demand (that is, the first time an object is requested by name). The message systems will be `ORAC::Msg::Control` objects (eg *ORAC::Msg::Control::AMS*).

This interface allows message systems to be initialised only when specific algorithm engines are required (rather than starting every message system even if none are required).

METHODS

The following methods are provided:

Constructor Object constructors.

new

Instantiate a new object ready for launching.

```
  $launch = new ORAC::Msg::MessysLaunch( $unique );
```

Since, in general, it is convenient for all parts of the code to have access to previously started message systems (and in many cases it is an error to start 2 identical message systems), the default behaviour is for the constructor to return the same object reference each time it is called. If it is required for a completely new object to be created each time the argument must be set to true.

ORAC-DR usually requires that access is provided to all previously initialised message systems so that the messaging layer can be configured by any subsystem.

Accessor Methods

config

Allows the message system configuration to be stored. These options are used to configure each message system that is initialised.

If it is required to configure message systems that are already running use the `configure_all` method.

Accepts a hash containing the names of the methods to invoke on the message system object and the options to use.

```
  $msl->config( messages => 1,
                timeout => 600,
                ... );
```

Currently, all options are configured at once and any previous options (even if they have different names) are lost.

Returns a hash with the current configuration.

messys

Retrieve the object associated with the specified message system, initialising it if required.

```
  $obj = $msl->messys("AMS");
```

`undef` is returned if the message system could not be initialised.

The message system object can be stored if two arguments are used. A rudimentary check is made to make sure that the object is a reference. It is not possible to check a true ISA relationship *until the class structure is reorganized*. If the object does not satisfy this condition it is not stored and a warning is raised with "-w".

```
  $msl->messys("AMS", $object);
```

Returns a hash reference containing all the currently launched engines if called without arguments.

```
  $launched = $launch->messys;
```

See also `messys_active`.

messys_active

Returns a hash containing all the message system objects that have been created.

```
 %Messys = $msl->messys_active;
```

preserve

This method is used to set or retrieve the `preserve` flag. The `preserve` flag controls whether the messys environment variables should be left unchanged for initialisation or whether the system should be initialised such that it does not interfere with non-ORAC-DR environments.

The default is that the message system should be initialised such that it does not interfere with other external systems. This is required if multiple ORAC-DR pipelines are to be run on the same machine by the same user.

If preserve is set to true it may be possible for the pipeline to interact with algorithm engines launched outside the context of the pipeline. This is the case when ORAC-DR is configure to interact with CGS4DR.

```
  $msl->preserve(1);
  $preserve = $msl->preserve;
```

General Methods

configure_all

Configure all the current message systems using the configuration options that have been set previously by use of the `config` method.

```
  $msl->configure_all;
```

configure_messys

Configures the named message system using the configuration options that have been set previously by use of the `config` method.

```
  $msl->configure_messys( 'AMS' );
```

init_messys

Given a message system name (for example 'AMS') initialise the message system so that it can be used by algorithm engines.

```
    $messys_obj = $msl->init_messys( 'AMS' );
```

Returns the object that was instantiated, or undef on error.

If the message system has been initialised previously that object is returned.

COPYRIGHT