Programming with Gap4

Version 0.99.2 (October 1997)

James Bonfield

Preface

This manual is a guide to programming with the newer Tcl/Tk based Staden Package programs. It covers both using the programs in a scripting environment and writing modules to extend the functionality of them. The main content current covers the Tcl interfaces, with very little of the C functions currently documented. The reader should also be familier with the Tcl language.

Conventions Used in This Manual

Italic is used for:

- variable names
- command line values
- structure fields

`Fixed width bold` is used for:

- Code examples
- Command line arguments
- Typed in commands
- Program output

The general format of the syntax for the more complex Tcl commands is to list the command name in bold followed by one or more command line arguments in bold with command line values in italic. The command line values have a brief description of the use of the value followed by the type and a default value. The Tcl convention of surrounding optional values in question marks is used. For instance the `edit_contig` command has the following syntax.

```
edit_contig
      -io           io_handle:integer
      -contig       identifier:string
     ?-reading      identifier:string()?
     ?-pos          position:integer(1)?
```

`-io` and `-pos` both take integer values. `-pos` is optional, and has a default value of 1. `-contig` and `-reading` both require string values. `-reading` is optional, and has a default value of a blank string.

1 Tk_utils Library

The *tk_utils* library provides basic Tcl and Tk extensions suitable for all applications. The common components of the programs, such as the text output display, keyed lists, and the configuration file handling are contained within this library.

The `stash` executable is a modified version of `wish` that contains these commands. When not using `stash` the Tcl load command must be used to dynamically link the library. From wish it is necessary to use the following startup code:

```
lappend auto_path $env(TKUTILS_LIBRARY)
catch {load libmisc.so}
catch {load libread.so}
load libtk_utils.so
```

The above assumes that the `TKUTILS_LIBRARY` and `LD_LIBRARY_PATH` environment variables have been set correctly. These are automatically done if the package initialisation files (‘`staden.profile`’ or ‘`staden.login`’) are sourced.

Once either `stash` or a boot-strapped `wish` is running the tk_utils library code is available.

1.1 Keyed Lists

Many functions make use of the TclX Keyed List extension. Keyed Lists can be compared to C structures. The following description has been taken from the TclX distribution[1].

<start of quotation>

A keyed list is a list in which each element contains a key and value pair. These element pairs are stored as lists themselves, where the key is the first element of the list, and the value is the second. The key-value pairs are referred to as fields. This is an example of a keyed list:

```
{{NAME {Frank Zappa}} {JOB {musician and composer}}}
```

If the variable *person* contained the above list, then `keylget person NAME` would return `{Frank Zappa}`. Executing the command:

```
keylset person ID 106
```

would make person contain

```
{{ID 106} {NAME {Frank Zappa}} {JOB {musician and composer}}
```

Fields may contain subfields; ‘.’ is the separator character. Subfields are actually fields where the value is another keyed list. Thus the following list has the top level fields `ID` and `NAME`, and subfields `NAME.FIRST` and `NAME.LAST`:

```
{ID 106} {NAME {{FIRST Frank} {LAST Zappa}}}
```

There is no limit to the recursive depth of subfields, allowing one to build complex data structures.

[1] The TclX copyright states the following.

Copyright 1992-1996 Karl Lehenbauer and Mark Diekhans. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies. Karl Lehenbauer and Mark Diekhans make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Keyed lists are constructed and accessed via a number of commands. All keyed list management commands take the name of the variable containing the keyed list as an argument (i.e. passed by reference), rather than passing the list directly.

`keyldel` *listvar key*
: Delete the field specified by key from the keyed list in the variable *listvar*. This removes both the key and the value from the keyed list.

`keylget` *listvar ?key? ?retvar* `|` `{}`*?*
: Return the value associated with key from the keyed list in the variable *listvar*. If *retvar* is not specified, then the value will be returned as the result of the command. In this case, if key is not found in the list, an error will result.

  If *retvar* is specified and key is in the list, then the value is returned in the variable retvar and the command returns 1 if the key was present within the list. If key isn't in the list, the command will return 0, and retvar will be left unchanged. If `{}` is specified for retvar, the value is not returned, allowing the Tcl programmer to determine if a key is present in a keyed list without setting a variable as a side-effect.

  If key is omitted, then a list of all the keys in the keyed list is returned.

`keylkeys` *listvar ?key?*
: Return the a list of the keys in the keyed list in the variable *listvar*. If keys is specified, then it is the name of a key field whose subfield keys are to be retrieve.

`keylset` *listvar key value ?key2 value2 ...?*
: Set the value associated with key, in the keyed list contained in the variable *listvar*, to value. If listvar does not exists, it is created. If *key* is not currently in the list, it will be added. If it already exists, *value* replaces the existing value. Multiple keywords and values may be specified, if desired.

<end of quotation>

An example best illustrates their usage. In this case we're using Gap4 to extract some *template* information for readings within an assembly database.

```
% set io [open_db -name TEST -version 1 -access rw]
% set r [io_read_reading $io 1]
% puts $r
{name 34} {trace_name 39} {trace_type 40} {left 25} {right 33} {position 90}
{length 545} {sense 1} {sequence 36} {confidence 37} {orig_positions 38}
{chemistry 0} {annotations 1} {sequence_length 440} {start 71} {end 512}
{template 1} {strand 0} {primer 1}
% set t [io_read_template $io [keylget r template]]
% puts $t
{name 45} {strands 1} {vector 1} {clone 1} {insert_length_min 1400}
{insert_length_max 2000}
% keylset t insert_length_max 2500
% puts $t
{name 45} {strands 1} {vector 1} {clone 1} {insert_length_min 1400}
```

```
    {insert_length_max 2500}
    % io_write_template $io [keylget r template] $t
    % close_db -io $io
```

The above is an interactive session. It starts by opening database `TEST`, version `1`. Then the first reading is loaded from the database and listed. Next the template for this reading is loaded and also listed. Finally, the maximum length for this template is changed to 2500 ,written back to the database, and the database closed.

1.2 Runtime Loading of Libraries

The main command for loading dynamic libraries is the `load_package` command. This adds on a new directory to the Tcl search path and dynamically loads up a new C library. For programmers, the procedure of creating these libraries is initially fairly complex. Once done, all the user requires is a single `load_package` command adding to the application '`rc`' file to extend the applications functionality.

The existing Tcl package system allows for the dynamic loading to be delayed until a command is needed. However this system does not satisfactorily deal with the case where libraries contain only C commands. Hence the package system utilised by the Staden Package dynamically links in libraries to the running executable at the time of the load_package call. This is typically done in the startup phase of programs.

---

load_package

load_package *name*

This loads the dynamic library named (eg) lib*name*.so. The "lib" and ".so" components of this library name a system dependent strings. The system will automatically use the correct local terminology depending on the system type.

Firstly the `$STADLIB/`*name* directory is appended to Tcl auto_path variable. Next the `$STADTABL/`*name*`rc` file is used to specify the package menus and defaults (which are saved as a keyed list in the global tcl variable *name*_defs). The `.`*name*`rc` file is also loaded up from the callers HOME directory and from the current directory, if they exist, in this order. This means that a user can override defaults specified in the `STADTABL` directory by creating an rc file in their home directory, and then to override these specifications further in a project-by-project fashion by adding configurations to the current directory.

Next the library itself is dynamically loaded. The file to be loaded is held within the `$STADLIB/$MACHINE-binaries` directory. If the library does not exist within this directory then it is not loaded and no error is produced.

Finally if existent, the package initialisation function in C will then be called with a Tcl interpreter as the sole argument and returns an integer (TCL_OK or TCL_ERROR). It is this function which performs the registering of new commands to the Tcl language. The C function name must be the package name with the first character as upper case, the following characters as lowercase, and suffixed by `_Init`. See the Tcl load manual page for full details.

So for the tk_utils library the `$STADLIB/tk_utils` directory is added to the auto_path variable, the `$STADTABL/tk_utilsrc` file is processed, and executes the C function `Tk_utils_Init()`.

---

load_package *tcldir libdir name ?init?*

This is the more versatile form of the load_package command. The procedures performed are the same, however the location of the files is no longer controlled solely by environment variables.

Tcldir specifies the directory to add to the Tcl auto_path variable and is used for the search path of the *name*`rc` file. As with the simpler form of load_package the `STADTABL`, HOME, and current directory versions of the rc file are also loaded, with each file overriding values specified in the earlier copies.

The *libdir* argument specifies the location to find the dynamic library file to load. Specifying this as a single `-` (minus sign) requests that no dynamic library is to be loaded. In this way libraries consisting solely of Tcl files may be used. Specifying *libdir* as a blank string (either `""` or `{}`) indicates that the library is to be searched for in the users `LD_LIBRARY_PATH` instead.

Both the *tcldir* and *libdir* variables allow a few substitutions to expand up to common locations.

%L	Expands to `$STADLIB`
%S	Expands to `$STADENROOT/src`
%%	Expands to a single percent sign

The *init* argument is used to indicate whether the dynamic library loaded has an initialisation routine. It should be set to 0 or 1. The current implementation always attemps to execute the initialisation routine, but when *init* is 0 errors will be ignored.

1.3 Default Files

The application *rc* files contain all the configuration details required by the application. Typically an *rc* file starts by loading up more packages using more `load_package` commands. This allows for hierarchial dependencies of packages and simplifies the loading of any single package. For instance, the '`siprc`' file contains a `load_package` call for seqlib. The '`seqlibrc`' file in turn has a `load_package` call for `tk_utils`.

Next we may define the menu data. Defining menus here allows for extensions to be written that add new commands directly onto the main menu. This obviously provides the ability to have third party extensions without sacrificing usability for the user. See Section 1.4 [Specifying Menu Configurations], page 7.

The rest of the *rc* file will contain the default value for applications. These may vary from the configuration parameters to the colours of plots to the text used in a particular dialogue. The available parameters to set are a function of the application itself, but the commands used to set these are universal.

---

set_def

set_def *parameter value*

This sets the application parameter *parameter* to *value*. *Parameter* is a Keyed List field within the application defs variable. If *value* is more than one word, the Tcl quoting mechanisms must be used. Valid examples are:

```
set_def CONSENSUS_CUTOFF                0.01
set_def STOP_CODON.RULER_COLOUR         black
set_def TRACE_DISPLAY.BACKGROUND        $normal_bg
set_def CONTIG_EDITOR.SE_SET.1          {0 0 0 1 0 0 0 0 1 1}
set_def CONTIG_EDITOR.SE_SET.1          "0 0 0 1 0 0 0 0 1 1"
```

The last two of these are different ways of acheiving the same result.

---

set_defx

set_defx *variable parameter value*

When we have common values to set for many parameters we may use the `set_defx` command. For example take the following settings:

```
set_def FIJ.HIDDEN.NAME         "Window size for good data scan"
set_def FIJ.HIDDEN.MIN          1
set_def FIJ.HIDDEN.MAX          200
set_def FIJ.HIDDEN.VALUE        100

set_def ASSEMBLE.HIDDEN.NAME    "Window size for good data scan"
set_def ASSEMBLE.HIDDEN.MIN     1
set_def ASSEMBLE.HIDDEN.MAX     200
set_def ASSEMBLE.HIDDEN.VALUE   100
```

The repetition here of common elements is tedious. Using `set_defx` the equivalent becomes:

```
set_defx defs_hidden    NAME    "Window size for good data scan"
set_defx defs_hidden    MIN     1
set_defx defs_hidden    MAX     200
set_defx defs_hidden    VALUE   100

set_def  FIJ.HIDDEN             $defs_hidden
set_def  ASSEMBLE.HIDDEN        $defs_hidden
```

1.4 Specifying Menu Configurations

By specifying menu configurations within the application rc file we provide the ability for extensions to include their own menu additions. When combined with the dynamic linking ability this means that new C functions can be written complete with GUI and menu items. These can then be "wrapped up" into a package suitable for distribution to other users.

Not all menus within our programs are specified within the configuration file, but typically the main menu is. Theoretically other menus (such as the gap4 contig editor ones) could be defined in this manner too.

An important concept in the menu code is menu states. At any time a menu item or a menu button can be either enabled or disabled (greyed out). Certain actions require a subset of the menu items to be enabled or disabled. Actions can be split into menu state changes that enable menu items and those that disable them. If an action needs to both enable and disable then two menu state changes should be applied. Menu states are specified as bit patterns with one bit per action.

For example in gap4 we have several enable states and several disable states.

bit	**Enable description**
0	Startup settings
2	A new database has been opened
3	The database has data

bit	**Disable description**

1	Busy mode has been set
2	The database has been closed
3	The database has no data
4	Read-only mode is enabled

Note that not all bits are used in the enable and disable settings. This is purely to simplify the numbering for the user. For example bits 2 and 3 have the same meaning for both the enable set and the disable set.

Bit 0 is always the startup setting. If a menu item does not have this bit set then it is disabled, otherwise it is enabled.

Bit 1 is always used by the busy mode. Busy mode disables items that have bit 1 set in the disable settings. When busy mode is turned off the menu settings revert to their initial state (prior to busy mode being enabled) and so no enable bit is necessary.

The other bits defined are application dependent. In this case bits 2 and 3 define whether the database opened and whether it contains data.

---

set_menu

set_menu *name*

The first menu command to be used is `set_menu`. This states that all further menu commands, until the next set_menu, will store their data in the Tcl variable *name*.

---

add_menu

add_menu *name onval offval pos*

Before adding commands to menus we need to create the menus themselves. The `add_menu` command does this. The menu *name* is the text to appear for the menu button. If this includes spaces it must be enclosed in quotes or curly brackets.

Onval and *offval* define the state masks for the enable and disable sets. Menus are always enabled whenever any of the items within them are enabled, even if the *offval* set defines otherwise. Menus are usually enabled at startup (*onval* == 1) and disabled during busy mode (*offval* == 2).

The *pos* argument may be either `left` or `right`. This requests the position to place the menu. Each leftwards positioned menu is packed to the right of the currently shown left menus. Hence the order in which menus are defined controls the order in which they will appear. Similarly for rightwards positioned menus.

So for example, the Gap4 main menus are defined as follows.

```
add_menu File           1 2 left
add_menu Edit           1 2 left
add_menu View           1 2 left
add_menu Options        1 2 left
add_menu Experiments    1 2 left
add_menu Lists          1 2 left
add_menu Assembly       1 2 left
add_menu Help           1 0 right
```

If more than one add_menu command is present for the same menu name the latter of the two takes priority.

add_cascade

add_cascade *name onval offval*

This adds a cascading menu item within an existing menu. The *name* should be the menu name followed by a full stop followed by the cascading menu name. So to add a `Save To` cascading menu to the `File` menu the *name* should be set to `"{File.Save To}"`.

Onval and *offval* operate in the same fashion as the `add_menu` command.

add_command

add_command *name onval offval command*

This adds a new command to an application. The *name* should be the menu pathname followed by fullstop followed by the name of the command to appear in the menu. So if the command is within a cascading menu the *name* will have several components broken down by fullstops, ending in the command name itself.

The *onval* and *offval* arguments control the states for which the command is to be enabled in.

The *command* argument is the command to execute when this menu item is selected. This is a single argument so Tcl quoting rules must be obeyed for multi-word commands. This command is evaluated (using the Tcl `eval` command) at the time of selecting the menu item. If the command is to contain references to variables, it is important to distinguish between variables expanded at the time of creating the menu item and the time of executing the menu item by backslashing the latter.

For example, the Gap4 "Quality" mode of the consensus output has the following specification.

```
    add_command  {File.Calculate a consensus.quality}  8 10  {QualityDialog \$io}
```

Here the "quality" command is within the "Calculate a consensus" cascading menu which is within the "File" menu. It is enabled by bit 3 (a database containing data has been opened) and is disabled by bits 1 and 3 (the database has no data or busy mode is enabled). The command to run is `QualityDialog $io`. If we did not backslash the `$io` in this command the *io* variable would be expanded up at the time of creating the menus, say to `"0"`. Then when the menu item is selected we would always execute `QualityDialog 0` which is not the desired effect.

add_separator

add_separator *name*

This simply adds a separator to the menu. The *name* specifies both the menu containing the separator and a name for the separator itself. Separator names do not appear in the menu, but are still required.

add_radio

add_radio *name onval offval variable value command*

Multiple radio buttons are grouped together to form a set of which any one button can be activated at any one time. The `add_radio` command adds commands to menus in a similar fashion to the `add_command` command, but has two additional arguments; *variable* and *value*.

Each radio button within a group uses the same *variable* with differing *values*. When a radio button is selected the global Tcl *variable* is set with the associated *value* and the *command* is executed. A useful tip is that the contents of the *variable* may be passed to the *command* as an argument using `\$`*variable*.

As each group can specify its own variable, multiple radio button groups are possible .

---

add_check

add_check *name onval offval variable command*

A check button command is identical to a normal command created by `add_command` except that the menu item also has a box showing the current toggled state. Unlike radio buttons each check button operates independently of every other check button.

The *variable* button specifies the global Tcl variable to hold the state for this check button. It will contain 1 for enabled and 0 for disabled. Whenever the item is selected the variable will be toggled and the command executed.

1.5 Controlling Menu Behaviour

The creation and control of menus within applications is governed by further menu commands. These do not appear within the configuration files but rather the Tcl code for the applications themselves.

---

create_menus

create_menus *menu_specs* ?*pathname*?

Uses the menu specifications passed over in the *menu_specs* variable to create the main menubar. *Pathname* specifies the root Tk window pathname in which to create the menus. If this is not specified the Tk root (.) is used instead.

The menu specifications are created from processing the application rc file. The `set_menu` command is used to specify a Tcl variable to store these specifications in. The contents of this variable should be used as the *menu_specs* argument.

---

menu_state_on

menu_state_on *menu_specs mask* ?*pathname*?

Enables menu items by applying menu state *mask* to the menus described in the *menu_specs* data. *Menu_specs* is the contents of the variable created by the `set_menu` command and written to by subsequent `add_`* commands.

The *mask* is applied to each item in the menu specs. If the menu item enable set ANDed with the *mask* is non zero the menu item is enabled. Otherwise it is not changed (and not disabled). It is possible to combine multiple enable bits together in a single call. Hence the following two examples are identical.

```
menu_state_on $gap_menu 4 .mainwin.menus
menu_state_on $gap_menu 8 .mainwin.menus

menu_state_on $gap_menu 12 .mainwin.menus
```

Pathname specifies the root location of the menu widgets as given to a previous `create_menus` command.

---

menu_state_off

menu_state_off *menu_specs mask ?pathname?*

This command is the same as the `menu_state_off` command except that menu items with their disable set value ANDed with the *mask* are disabled. Otherwise the menu item is left in the current state.

---

menu_state_set

menu_state_set *menu_spec_variable mask ?pathname?*

This command provides a combined interface to the `menu_state_on` and `menu_state_off` functions. The name of the global variable containing the menu specifications is passed over in the *menu_spec_variable* argument. This must have been set by using the `set_menu` command.

If the *mask* value is positive the `menu_state_on` command is called with this mask, otherwise the `menu_state_off` command is called with the absolute value of the *mask*.

Pathname specifies the root location of the menu widgets as given to a previous `create_menus` command.

---

menu_state_save

menu_state_save *pathname*

This command queries the current states of the menus created as children of *pathname* and returns them as a string suitable for passing to a later `menu_state_restore` function. The principle use of this function is within the `SetBusy` command.

---

menu_state_restore

menu_state_restore *pathname states*

This commands sets the current states of the menus created as children of *pathname*. The *state* variable contains the menu state information as returned from an earlier *menu_state_save* function. The principle use of this function is within the `ClearBusy` command.

1.6 Common Dialogue Components

This section has yet to be written. I need to outline the basic tk_utils widget-like commands: radiolist, entrybox, checklist, okcancelhelp, ColourBox, repeater, scalebox and yes_no. The interfaces will probably change to tidy things up before this section is written.

okcancelhelp

checklist

entrybox

messagebox

radiolist

renzbox

repeater

scalebox

scale_range

yes_no

1.7 Text Output and Errors

A selection of C and Tcl functions exist for outputting text to either the stdout or stderr streams. For entirely text based applications these messages simply appear on their usual streams. For graphical applications the messages can appear in the main window of the application. The programmer is free to use the usual C output routines, such as `printf`, but doing so will no output to the main window.

To utilise the text based version of the routines no initialisation is required. For the windowing version the following startup code should be used from within `stash`.

```
tkinit
pack [frame .output -relief raised -bd 2] -fill both -expand 1
load_package tk_utils
tout_create_wins .output
```

In the above example `.output` can be replaced by any window name you choose. The `load_package tk_utils` command is required to load the *tk_utils_defs* variable. The `tout_create_wins` command does the actual work of creating the necessary output and error windows. From then on, the text output routines will send data to the windows instead of stdout and stderr.

---

tout_init

`tout_init` *output_path error_path*

This command initialises the redirection of the text output commands. The two rrquired arguments specify the Tk pathnames of text widgets for the output and errors to be sent to. The function returns nothing.

The following example illustrates the usage. In practise the `tout_create_wins` command should be used instead to provide a common style interface.

```
pack [text .output -height 5] [text .error -height 5] -side top
tout_init .output .error
vmessage This appears in the output window
verror ERR_WARN This appears in the error window
```

---

tout_create_wins

`tout_create_wins` *frame*

This creates output and error windows within the specified *frame*. *frame* may be `{}` to add these directly to the top level. The function returns nothing. The windows created also contain functional search, scroll on output, clear, and redirect buttons.

This function also calls the `tout_init` command to initialise redirection of the text output functions.

---

tout_set_scroll

`tout_set_scroll` *stream to_scroll*

This command controls whether outputting text should automatically scroll the relevant output window so that the new text is visible. *stream* should be one of `stdout` or `stderr`. If *to_scroll* is 0, scrolling is not automatically performed, otherwise scrolling is performed.

This control is connected to the "scroll on output" button created by the `tout_create_wins` command.

tout_set_redir

`tout_set_redir` *stream filename*

This command can be used to enable redirection of any output or error to a file. Output also still appears in the appropriate window. *stream* should be one of `stdout` or `stderr`. *filename* specifies which file to save output to. Any previously redirected filename for this stream is automatically closed. A blank *filename* can be used to close the current redirection for this stream without opening a new file. The command returns 1 for success, 0 for failure.

This control is connected to the `"redirect"` menu created by the `tout_create_wins` command.

tout_pipe

`tout_pipe` *command input forever*

This command executes the unix shell *command* with *input*. If *forever* is 0, the command is terminated if it takes more than a specific amount of time (currently 5 seconds). A value of *forever* other than 0 causes the `tout_pipe` command to wait until *command* has finished. The stdout and stderr streams from *command* appear in the appropriate output window. The command returns 0 for success, -1 for failure.

NOTE: This command may not be implemented on all platforms.

error_bell

`error_bell` *status*

This command controls whether a bell should be emitted for each error displayed. (Currently bells only ring for the C implementation of `verror` and not the Tcl one). If *status* is 0, no bell is rung.

vmessage

```
#include <text_output.h>
void vmessage(char *fmt, ...);
```

This C function displays text in the text output window or prints to stdout when in a non graphical environment. Arguments are passed in the standard `printf` syntax. Hence `vmessage("output");` and `vmessage("value=%d",i);` are both legal uses.

`vmessage` ?*text ...*?

This is the Tcl interface to the vmessage C function. Any number of arguments can be specified. Each are concatenated together with spaces inbetween them.

verror

```
#include <text_output.h>
void verror(int priority, char *name, char *fmt, ...);
```

This C function displays text in the error output window or prints to stderr when in a non graphical environment. The *priority* argument may be one of `ERR_WARN` or `ERR_FATAL`. The *name* argument is used as part of the error message, along with the time stamp and the error itself. *name* should not be any more than 50 characters long, and ideally much shorter. The *fmt* arguments onwards form the standard `printf` style arguments of a format specifier and string components.

An error with priority of `ERR_WARN` will be sent only to the error window. Priority `ERR_FATAL` will print to stderr as well. `ERR_FATAL` should be used in conditions where there is a chance that the program may subsequently crash, thus removing the error window from the screen and preventing users from reporting error messages.

`vmessage` *priority text* ?...?

This is the Tcl interface to the verror C function. The *priority* argument should be one of `ERR_WARN` or `ERR_FATAL` as described above. The *text* and subsequent arguments make up the contents of the error message itself with each argument concatenated with a single space between arguments. The Tcl (not C) implementation of `verror` currently has a limit of 8192 bytes of error message per call.

---

vfuncheader

```
#include <text_output.h>
void vfuncheader(char *fmt, ...);
```

This C function displays the name of a function in the output window. The function header consists of ruler lines, the date and time, and the formatted string specified by the *fmt* and subsequent arguments. These arguments should be specified in the standard `printf` style. The header, after formatting, must be less than 8192 bytes long.

`vfuncheader` *title*

This is the Tcl interface to the vfuncheader C function. It takes a single argument named *title* and uses this as the function title. The *title* must be less 8192 bytes long.

---

vfuncgroup

```
#include <text_output.h>
void vfuncgroup(int group, char *fmt, ...);
```

This C function is identical to the `vfuncheader` function except that it will not output a new header if the last call to `vfuncgroup` was with the same *group* number and there have been no intevening `vfuncheader` calls.

The *group* argument is an integer value specifying a group number. Each option within a program using this function should have its own unique group number. However currently there is no allocation system for ensuring that this is so. The *fmt* and subsequent arguments specify the header in the standard `printf` style. The header, after formatting, must be less than 8192 bytes long.

`vfuncgroup` *group_number title*

This is the Tcl interface to the vfuncheader C function. The *title* must be less 8192 bytes long.

---

vfuncparams

```
#include <text_output.h>
void vfuncparams(char *fmt, ...);
```

This function sets the parameters used for producing the current output. These are added as a tagged text segment to the text underneath the last displayed header. The right mouse button in the output window brings up a menu from which these parameters can be displayed. By default they are not displayed. The parameters can be any length and are specified by *fmt* and subsequent arguments in the standard `printf` style.

---

start_message and end_message

```
#include <text_output.h>
void start_message(void);
void end_message(void);
```

Sometimes we wish to bring up a separate window containing simple message outputs (eg in gap4 this could be information about a reading that was clicked on). The `start_message` function clears the current message buffer and starts copying all subsequent output to the stdout window to this buffer.

The `end_message` function disables this message copying and display the current contents of the message buffer in a separate window.

At present, there are no Tcl interface to these routines.

1.8 Other Utility Commands

tkinit

`tkinit`

This command calls the `Tk_Init` C function. The purpose of this function is to allow the `stash` program to be used in a non windowing environment. To achieve this the initialisation of Tk has been delayed until this command is ran. Hence one binary can be used for both text work (no `tkinit` call) and graphics work (with a `tkinit` call).

capture

`capture` *command* ?*varname*?

This command executes *command* and stores any text written to stdout in the tcl variable named in *varname*. If *varname* is not specified then the output is returned, otherwise the return codes from the `Tcl_Eval` routine are used (ie `TCL_OK` for success).

For example the command `"set x [capture {puts foo}]"` and `"capture {puts foo} x"` both set *x* to contain "*foo\n*".

---

expandpath

`expandpath` *pathname*

This command returns an expanded copy of *pathname* with tilde sequences and environment variables expanded in a usual shell-like fashion. It is a direct interface to the `expandpath` C routine, so see this for full details.

For example, the command `"expandpath {~/bin/$MACHINE}"` may return a string like "*/home5/pubseq/bin/alpha*".

---

vTcl_SetResult

```
#include <tcl_utils.h>
void vTcl_SetResult(Tcl_Interp *interp, char *fmt, ...);
```

This function is a varargs implementation of the standard `Tcl_SetResult` function. The Tcl result is set to be the string specified by the *fmt* and subsequent arguments in the standard `sprintf` style.

NOTE: The current implementation has a limit of setting up to 8192 bytes.

---

vTcl_DStringAppend

```
#include <tcl_utils.h>
void vTcl_DStringAppend(Tcl_DString *dsPtr, char *fmt, ...);
```

This function is a varargs implementation of the standard `Tcl_DStringAppend` function. The string specified by the *fmt* and subsequent arguments (in the standard `sprintf` style) is appended to the existing dynamic string.

---

w and vw

```
#include <tcl_utils.h>
char *w(char *str);
char *vw(char *fmt, ...);
```

These functions return strings held in writable memory. Writable strings are required in the arguments of many Tcl functions, including `Tcl_SetVar` and `Tcl_GetKeyedListField`. The arguments specify the string the return as writable. For `w()` this is simply an exact copy of the argument. For `vw()` the returned string is a formatted copy of the input, which is specified in the standard `printf` style.

The return value from the `w` function isvalid only until the next call of `w()`. Similarly for the `vw` function.

Examples of usage are:

```
Tcl_GetKeyedListField(interp, vw("MODE%d", mode_num), gap_defs, &buf);

Tcl_SetVar(interp, w("arr(element)"), "10", TCL_GLOBAL_ONLY);
```

NOTE: In the current implementations both functions have a limit of handling 8192 bytes per call.

2 Tcl Scripting of Gap4

2.1 Introduction

This chapter describes the gap4 scripting language. The language is an extension of the Tcl and Tk languages. This manual does not contain information on using Tcl and Tk itself - only our extensions.

For the purpose of consistency, many gap4 commands take identical arguments. To simplify the documentation and to remove redundancy these arguments are only briefly discussed with each command description. However first we need to describe the terminology used throughout this manual.

Reading identifier
: Used to specify a reading. It can consist of the reading's unique name, a hash followed by its reading number, or if it is at the start of a contig, an equals followed by the contig number.

  Eg `fred.s1`, `#12`, or `=2`.

Contig identifier
: A contig is identified by any reading within it, so all reading identifiers are contig identifiers. However when a contig identifier is displayed by a command it typically chooses the left most reading name. If a contig number is known, simply use `=`*number* as a contig identifier.

Common arguments:

`-contigs` *contig_list*
: *Contig_list* is a Tcl list of contig identifiers. If an item in the list is itself a list, then the first element of the list is the identifier and the second and third elements specify a range within that contig.

  Eg `-contigs {read1 {read5 1000 2000} read6}`

`-readings` *reading_list*
: *Reading_list* is a Tcl list of reading identifiers.

  Eg `-reading_list {read1 read2}`

`-contig` *contig_identifier*
: Specifies a single contig by an indentifier.

`-reading` *reading_identifier*
: Specifies a single reading by an identifier.

`-cnum` *contig_number*
: Specifies a contig by its number (NB: this not the same as a reading number within that contig).

`-rnum` *reading_number*
: Specifies a reading by its number.

`-io` *io_handle*
: Specifies an IO handle by a numerical value as returned from a previous `open_db` command.

2.2 Low-level IO Access

2.2.1 Introduction

FIXME: Add intro here

2.2.2 IO Primitives

2.2.2.1 io_read_text and io_write_text

The database structures typically contain record numbers of text strings rather than copies of the strings themselves, as this easily allows resizing of the strings.

`io_read_text` *io record_number*
: Reads the text from *record_number* and returns it. Results in a Tcl error if it fails.

`io_write_text` *io record_number text*
: Writes *text* to the requested *record_number*. Returns 0 for success, -1 for failure

2.2.2.2 io_read_data and io_write_data

These functions are for reading and writing the binary data in the database.

2.2.2.3 Flushing data

When updating the database information it is often necessary to perform several edits. Initially we assume that the database is consistent and correct. After updating the database we also wish for the database to be consistent and correct. However during update this may not be true.

Consider the case of adding a new reading to the end of a contig. We need to write the new reading with it's left neighbour set to the original last reading in the contig; then need to update the original last reading's right neighbour to reference our new last reading; and finally we need to update the contig information. During this operation the database is inconsistent so should the program or system terminate unexpectedly we wish to revert back to the earlier consistent state.

This is performed by use of controlled flushing. The database internally maintains a time stamp of the last flushed state. When we open a database that contains data written after the last flush we ignore the new data and use the data written at the last flush.

`io_flush` *io*
: A Tcl function to flush the data stored in the database reference by *io*. Always returns success.

`void flush2t(`*GapIO *io*`)`
: A C function to flush the database. Void return type.

`gap_auto_flush`
: A variable to control whether the Tcl level io write commands (eg. `io_write_reading`, `io_write_reading_name`, `io_add_reading` and `io_allocate`) automatically flush after performing the write. Note the consequences of this action. By default this is set to 0 which disables automatic flush. A non zero value enables automatic flush.

2.2.3 Low-level IO Commands

2.2.3.1 Opening, Closing and Copying Databases

Before any database accessing can take place the gap4 database must be opened. This is done using the `open_db` call. This returns an *io* handle which should be passed to all other functions accessing the database.

```
open_db
      -name         database_name
     ?-version      version?
     ?-create       boolean?
     ?-access       access_mode?
```

This opens a database named *database_name*. The actual files used will be *database_name.version*. The routine is used for both creating a new database and opening an existing database. The value returned is the io handle of the opened database. More than one database may be opened at one time.

-name *database_name*
: Specifies the database name. The name is the start component of the two filenames used for storing the database and so is the section up to, but not including, the full stop. This is not an optional argument.

-version *version*
: This optional parameter specifies the database version. The version is the single character after the full stop in the UNIX database filenames. It is expected to be a single character. The default value (as used for newly created databases) is `"0"`.

-create *boolean*
: Whether to open an existing database (-create 0) or a new database (-create 1). The default here is 0; to open an existing database.

-access *access_mode*
: The *access_mode* specifies whether the database is to be opened in read-only mode or read-write mode. Valid arguments are "r", "READONLY", "rw" and "WRITE". If a database is opened in "rw" or "WRITE" mode a BUSY file will be created. If the BUSY file already exists then the database is opened in read-only mode instead. Either way, the `read_only` Tcl variable is set to 0 for read-write and 1 for read-only mode.

```
close_db
      -io           io_handle
```

This closes a previously opened database. Returns nothing, but produces a Tcl error for failure.

-io *io*
: Specifies which database to close. The *io* is the io handle returned from a previous `open_db` call. Attempting to close databases that have not been opened will lead to undefined results.

```
copy_db
      -io           io_handle
      -version      version
     ?-collect      boolean?
```

This command copies a currently open database to a new version number. The currently opened database is not modified. After copying the current open database referred to by *io_handle* is still the original database.

-io *io*
: Specifies which database to copy. The *io* is the io handle returned from a previous `open_db` call. Attempting to copy databases that have not been opened will lead to undefined results.

-version *version*
: This parameter specifies the database version to create to place the copy in.

-collect *boolean*
: This optional parameter specifies whether to perform garbage collection when copying the file. A value of 0 means no garbage collection; which is simply to do a raw byte-by-byte copy of the two database files. A non zero value will read and write each reading, contig, (etc) in turn to the new database, thus resolving any database fragmentation. The default value is "0".

2.2.3.2 io_read_database

The database structure holds information that is relevant to the entire project rather than on a per reading, per contig or per 'whatever' basis. Among other things it keeps track of the amount of information stored.

`io_read_database` *io*
: Reads the database structure from a specified *io* number and stores it in a keyed list. Returns the structure as keyed list when successful, or a blank string for failure.

`io_write_database` *io keyed_list_contents*
: Writes the database structure stored in the *keyed_list* to a specified *io* number. Returns 0 for success, -1 for failure.

For a description of the database structure, see (FIXME) "The GDatabase Structure".

2.2.3.3 io_read_reading

The reading structure holds the primary information stored for each sequence. It references several other structures by their numbers into their own structure index. The reading structures also contain references to other reading structures. This is done by use of a doubly linked list ("left" and "right" fields), sorted on ascending position within the contig.

`io_read_reading` *io reading_number*
: Reads a reading structure from a specified *io* number and stores it in a keyed list.

`io_write_reading` *io reading_number keyed_list_contents*
: Writes a reading structure stored in the *keyed_list* to a specified *io* number. Returns 0 for success, -1 for failure.

For a description of the readinf structure, see (FIXME) "The GReadings Structure".

2.2.3.4 io_read_contig

The contig structure holds simple information about each contiguous stretch of sequence. The actual contents and sequence of the contig is held within the readings structures, including the relative positioning of each sequence.

`io_read_contig` *io contig_number*
: Reads a contig structure from a specified *io* number and stores it in a keyed list.

`io_write_contig` *io contig_number keyed_list_contents*
: Writes a contig structure stored in the *keyed_list* to a specified *io* number. Returns 0 for success, -1 for failure.

For a description of the database structure, see (FIXME) "The GContigs Structure".

2.2.3.5 io_read_annotation

Annotations, also known as tags, are general comments attached to segments of sequences (either real readings or the consensus). They form a singly linked list by use of the "next" field. The annotations must be sorted in ascending order.

`io_read_annotation` *io annotation_number*
: Reads an annotation structure from a specified *io* number and stores it in a keyed list.

`io_write_annotation` *io annotation_number*
: *keyed_list_contents* Writes an annotation structure stored in the *keyed_list* to a specified *io* number. Returns 0 for success, -1 for failure.

For a description of the database structure, see (FIXME) "The GAnnotations Structure".

2.2.3.6 io_read_vector

This holds information used on the vectors (one structure per vector) used for all stages of cloning and subcloning. For example both m13mp18 and pYAC4 vectors.

`io_read_vector` *io vector_number*
: Reads a vector structure from a specified *io* number and stores it in a keyed list.

`io_write_vector` *io vector_number keyed_list_contents*
: Writes a vector structure stored in the *keyed_list* to a specified *io* number. Returns 0 for success, -1 for failure.

For a description of the database structure, see (FIXME) "The GVectors Structure".

2.2.3.7 io_read_template

The template is the final piece of material used for the readings. So if we sequenced the insert from both ends then we would expect to have two reading structures referencing this template structure.

`io_read_template` *io template_number*
: Reads a template structure from a specified *io* number and stores it in a keyed list.

`io_write_template` *io template_number keyed_list_contents*
: Writes a template structure stored in the *keyed_list* to a specified *io* number. Returns 0 for success, -1 for failure.

For a description of the database structure, see (FIXME) "The GTemplates Structure".

2.2.3.8 io_read_clone

The clone is the the material that our templates were derived from. Typically the clone name is used as the database name. Example vectors are cosmid, YAC or BAC vectors.

`io_read_clone` *io clone_number*
: Reads a clone structure from a specified *io* number and stores it in a keyed list.

`io_write_clone` *io clone_number keyed_list_contents*
: Writes a clone structure stored in the *keyed_list* to a specified *io* number. Returns 0 for success, -1 for failure.

For a description of the database structure, see (FIXME) "The GClones Structure".

2.2.3.9 io_read_reading_name and io_write_reading_name

When accessing the reading name record referenced by the reading structure, special purpose functions must be used. The reading names are cached in memory once a database is opened. This speeds up accesses, but requires different IO functions. Note that `io_write_text` to update a reading name will invalidate the cache and cause bugs.

`io_read_reading_name` *io reading_number*
: Returns the reading name for reading *reading_number*.

`io_write_reading_name` *io reading_number name*
: Writes the new reading *name* for reading *reading_number*. Assuming correct syntax, this always returns success.

2.2.3.10 io_add_* commands and io_allocate

A set of Tcl functions exists for allocating new gap4 database structures. Each function allocates the next sequentially numbered structure. In the case of annotations it is preferable to reuse items stored on the annotation free list before allocating new structures.

`io_add_reading` *io*
: Creates a new reading numbered `NumReadings(io)+1`. The name, trace_name and trace_type fields are all allocated and written as "uninitialised". No other items are allocated and all other fields are set to 0. The database num_readings and Nreadings fields are also updated. Returns the new reading number.

`io_add_contig` *io*
: Creates a new contig numbered `NumContigs(io)+1`. The contig structure fields are all set to 0. The contig_order array is updated with the new contig as the last (right most) one. The database num_contigs and Ncontigs fields are also updated. Returns the new contig number.

`io_add_annotation` *io*
: Creates a new annotation. The structure fields initialised to 0. The database Nannotations field is also updated. Returns the new annotation number.

`io_add_template` *io*
: Creates a new template. The template name is allocated and set to "uninitialised"; strand is set to 1; vector is set to the "unknown" vector (1) if present or creates a new blank vector to reference; and the clone, insert_size_min and insert_size_max

are set to 0. The database Ntemplates field is also updated. Returns the new template number.

`io_add_vector` *io*
: Creates a new vector. The vector name is allocated and set to `"uninitialised"`. The level is set to 0. The database Nvectors field is also updated. Returns the new vector number.

`io_add_clone` *io*
: Creates a new clone. The clone name is allocated and set to `"uninitialised"`. The vector is set to the `"unknown"` vector(1) or creates a new blank vector to reference. The database Nclones field is also updated. Returns the new template number.

`io_allocate` *io type*
: Allocates a new record of the specified *type*. Currently only the `text` type is supported. The new record number is returned.

2.3 Utility Commands

db_info

`db_info num_readings` *io*

This command returns the number of readings in the database.

`db_info num_contigs` *io*

This command returns the number of contigs in the database.

`db_info t_contig_length` *io*

This command returns the total number of characters in the consensus for all contigs.

`db_info t_read_length` *io*

This command returns the total number of bases used in all the readings.

`db_info get_read_num` *io reading_identifier*

This command returns the reading number (between 1 and *num_contigs*) for a specific reading. For instance, to convert the reading name `xb64a10.s1` to its reading number we use:

```
set rnum [db_info get_read_num $io xb64a10.s1]
```

If the reading name is not found, -1 is returned.

`db_info get_contig_num` *io reading_identifier*

This command returns the contig number for a specific reading. The number returned is the number of the contig structure, not the number of the left most reading within this contig. It returns -1 for failure.

`db_info chain_left` *io reading_identifier*

This command returns the left most reading number within a contig specified by the *reading_identifier*. It returns -1 for failure.

`db_info longest_contig` *io*

This command returns the contig number (not the left most reading number) of the longest contig in the database.

`db_info db_name` *io*

This command returns the name of the database opened with the specified *io* handle. The name returned includes the version number, so an example result would be `TEST.0`.

---

edid_to_editor

`edid_to_editor` *editor_id*

This command converts the contig editor identifier number to the Tk pathname of the associated Editor widget. The contig editor identifier can be obtained from acknowledging (within C) a REG_CURSOR_NOTIFY event.

---

add_tags

```
add_tags
      -io            io_handle:integer
      -tags          tag_list:strings
```

This command adds a series of annotations to readings and contigs within the database.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-tags` *tag_list*
: This specifies the list of annotations to add. The format of *tag_list* is as a Tcl list of tag items, each of the format:

  reading_number tag_type direction start`..`*end comment_lines*

  If the *reading_number* is negative the tag is added to the consensus of the contig numbered -*reading_number*. The *tag_type* should be the four character tag type. The *direction* should be one of "+", "-" or "=" (both). The *start* and *end* specify the inclusive range of bases the annotation covers. These count from 1 in the original orientation of the sequence. The *comment_lines* hold the text for the annotation. Several lines may be included.

The following example adds two tags. The first is to reading #12 from position 10 to 20 inclusive. The second is to contig #1.

```
     set t "{12 COMM + 10..20 comment} {-1 REPT = 22..23 multi-line\ncomments}"
     add_tags -io $io -tag_list $t
```

---

get_read_names

```
get_read_names
      -io            io_handle:integer
     ?identifier ...?
```

This command converts a list of reading identifiers to reading names. The identifiers can be either "#number" or the actual read name itself, although the command is obviously only useful for the first syntax. The names are returned as a Tcl list.

---

contig_order_to_number

contig_order_to_number

```
      -io            io_handle:integer
      -order         position:integer
```

This command converts a contig position number to a contig number. That is we can ask "which is the second contig from the left". The function returns the coontig number.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-order` *position*
: The position of the contig. "1" is the left most contig.

reset_contig_order

reset_contig_order

```
    -io           io_handle:integer
```

This command resets the contig order so that the lowest numbered contig is at the left and the highest numbered contig at the right. The new contig order is written to disk and the database is flushed.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

flush_contig_order

flush_contig_order

```
    -io           io_handle:integer
```

This command writes the contig order information to disk and then runs the `io_flush` command.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

remove_contig_duplicates

remove_contig_duplicates

```
    -io           io_handle:integer
    -contigs      identifiers:strings
```

This function removes duplicate contig identifiers from a given list. The function takes a list of identifiers (in the usual name or #number fashion) and returns a list of the left most reading names in the contigs. If two different identifiers for the same contig are given, only the one identifier is returned.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: The list of contig identifiers.

get_tag_array

get_tag_array

This function parses the tag databases and returns a Tcl list containing the tag information. Each element of the returned list consist of the tag name, its type, and the default comment.

For instance, the standard installation returns a list starting with `"{comment COMM ?} {oligo OLIG {}} {compression COMP {}} {stop STOP {}}"`.

2.4 Main Commands

assemble_direct

```
assemble_direct
      -io             io_handle:integer
      -files          filenames:strings
     ?-output_mode    mode:integer(0)?
     ?-max_pmismatch  percentage:float(-1)?
```

This performs the gap4 directed assembly function. It takes a list of Experiment File filenames and processes these according to their content. The Experiment Files should contain AP lines to govern their positions in the assembly.

The function returns a list of failed files separated by newlines.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-files` *filenames*
: A Tcl list of Experiment File filenames.

`-output_mode` *mode*
: Whether to display alignments when assembling sequences containing a tolerance of zero or more. A *mode* of non-zero displays alignments, 0 does not. This is an optional argument with the default as 0.

`-max_pmismatch` *percentage*
: When aligning sequences (tolerance >= 0) the aligned sequences must match the consensus sequence with <= *percentage* mismatch. A *percentage* of -1 implies no check should be made. The default for this option is -1.

---

assemble... commands

```
assemble_independent
      -io              io_handle:integer
      -files           filenames:strings
     ?-output_mode     mode:integer(1)?
     ?-min_match       length:integer(20)?
     ?-min_overlap     length:integer(0)?
     ?-max_pads        count:integer(25)?
     ?-max_pmismatch   percentage:float(5.0)?
     ?-joins           to_join:integer(1)?
     ?-enter_failures  to_enter:integer(0)?
     ?-tag_types       types:strings()?

assemble_new_contigs
      -io              io_handle:integer
      -files           filenames:strings

assemble_one_contig
      -io              io_handle:integer
      -files           filenames:strings
```

```
assemble_screen
      -io             io_handle:integer
      -files          filenames:strings
     ?-output_mode    mode:integer(1)?
     ?-min_match      length:integer(20)?
     ?-min_overlap    length:integer(0)?
     ?-max_pads       count:integer(25)?
     ?-max_pmismatch  percentage:float(5.0)?
     ?-save_align     to_save:integer(0)?
     ?-win_size       length:integer(0)?
     ?-max_dashes     count:integer(0)?
     ?-tag_types      types:strings()?
```

```
assemble_shotgun
      -io             io_handle:integer
      -files          filenames:strings
     ?-output_mode    mode:integer(1)?
     ?-min_match      length:integer(20)?
     ?-min_overlap    length:integer(0)?
     ?-max_pads       count:integer(25)?
     ?-max_pmismatch  percentage:float(5.0)?
     ?-joins          to_join:integer(1)?
     ?-enter_failures to_enter:integer(0)?
     ?-tag_types      types:strings()?
```

```
assemble_single_strand
      -io             io_handle:integer
      -files          filenames:strings
     ?-output_mode    mode:integer(1)?
     ?-min_match      length:integer(20)?
     ?-min_overlap    length:integer(0)?
     ?-max_pads       count:integer(25)?
     ?-max_pmismatch  percentage:float(5.0)?
     ?-joins          to_join:integer(1)?
     ?-enter_failures to_enter:integer(0)?
```

The assembly functions listed above all take similar arguments, but perform varying modes of assembly. The complete list of available arguments is listed below, but note that not all arguments apply to each function. Most functions return the failed readings and error codes with newlines between each reading and error code pair. `Screen_only` may return (when *save_align* is enabled) the reading alignment scores in a similar fashion.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-files` *filenames*
: *Filenames* must contain a Tcl list of files to assemble.

`-output_mode` *mode*
: Specifies the level of verbosity of the output. The default is 0. *Mode* must be one of the following.

1	Display no alignments
2	Display only passed alignments

3	Display all alignments
4	Display only failed alignments

`-min_match` *length*
: Specifies the minimum length of exact match used during the hashing stage of assembly. The minium allowed value for this is 8. The default is 20.

`-min_overlap` *length*
: This specifies the minimum length of an overlap between a reading and a consensus sequence. The default is 0 which implies no overlap is too short. Note that `-min_match` is still used so all overlaps have to be larger than that parameter in order to be found.

`-max_pads` *count*
: After alignments the number of pads required in each of the two sequences (consensus and reading, or two consensuses) must be less than or equal to *count*. The default is 25.

`-max_pmismatch` *percentage*
: After alignments the percentage of bases that do not match must be less than or equal to *percentage*. This is a floating point value. The default is 5.0.

`-save_align` *to_save*
: This controls whether alignments scores are to be returned as the function result. A non zero value returns the scores. The default is 0.

`-win_size` *length*
`-max_dashes` *count*
: During a screen-only assembly the cutoff data may be searched for matches. The criteria for determining how much cutoff sequence to align is selected as the portion where no more than *count* unknown (`"-"`) bases within a region of *length* bases. Setting both these parameters to 0 means that cutoff data will be not searched. These are the defaults.

`-joins` *to_join*
: This controls whether joins between contigs should be allowed. A non zero value allows joins. The default is 1.

`-enter_failures` *to_enter*
: This controls whether failed readings should still be entered into the databases as single reading contigs. A non zero value will enable this. The default is 0.

`-tag_types` *types*
: The assembly algorithm can mask segments of sequence covered by tags so that they are not used during hashing step and hence do not initiate overlaps. If *types* is a non blank list of tag types then masking will be applied to hide sequence covered by tags of these types from the initial hashing stage of assembly. The default is a blank list, which means no masking will be performed.

break_contig

```
break_contig
      -io           io_handle:integer
      -readings     identifiers:strings
```

This command breaks contigs into two or more pieces at given points. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-readings` *identifiers*
: This specifies the list of readings. For each reading the contig will be broken such that the reading forms the left end of a new contig.

TODO: calc_quality

```
calc_quality
      -io           io_handle:integer
      -contig       contig
      -window       window
```

This command will have the interface updated in the future to conform to the style used by other commands. Use at your own risk.

---

check_assembly

```
check_assembly
      -io           io_handle:integer
      -contigs      identifiers:strings
     ?-cutoff       use_cutoffs:integer(1)?
     ?-min_len      length:integer(10)?
     ?-win_size     length:integer(29)?
     ?-max_dashes   count:integer(3)?
     ?-max_pmismatch percentage:float(15.0)?
```

The command performs the Gap4 Check Assembly command. It compares either the used data or the cutoff data against the consensus sequence to find readings with poor match. The function is not currently ideally suitable for use in a script as it plots directly to the Contig Selector display. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to check. Only the contig identifier is currently used, although the syntax specifying start and end ranges is valid.

`-cutoff` *use_cutoffs*
: Controls whether the cutoff data is to be analysed. If *use_cutoffs* is a non zero value the cutoff data will be aligned and compared against the consensus. Otherwise the already aligned used data will be compared against the consensus. The default is 1.

`-min_len` *min_length*
`-win_size` *window_length*
`-max_dashes` *count*
: These parameters are only used when *-cutoff* is enabled. The criteria for determining how much cutoff sequence to align is selected as only the portion where no more than *count* unknown (`"-"`) bases within a region of *window_length* bases. This sequence is then only used if the amount selected is at least *min_length* bases long. The defaults are 10 for `-min_len`, 29 for `-win_size` and 3 for `-max_dashes`.

`-max_pmismatch` *percentage*
: Only matches with greater than the specified percentage mismatch are displayed as problems. The default is 15.0.

---

check_database

```
check_database
        -io             io_handle:integer
```

This function performs the gap4 check database function. It returns the number of serious database corruptions detected.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

---

complement_contig

```
complement_contig
        -io             io_handle:integer
        -contigs        identifiers:strings
```

This command complements one or more contigs and writes back the modified data to the database. It returns 0 for success and 1 for failure.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to complement. Only the contig identifier is currently used, although the syntax specifying start and end ranges is valid.

---

delete_contig

```
delete_contig
        -io             io_handle:integer
        -contigs        identifiers:strings
```

This command deletes one or more contigs from the database, including readings and associated information. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to delete. Only the contig identifier is currently used, although the syntax specifying start and end ranges is valid.

---

disassemble_readings

```
disassemble_readings
      -io            io_handle:integer
      -readings      identifiers:strings
     ?-all           for_all:integer(1)?
     ?-remove        to_remove:integer(1)?
```

This command disassembles readings by either removing them from the database or moving them to their own contigs. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-readings` *identifiers*
: Specifies the list of readings to disassemble.

`-all` *for_all*
: Controls whether to disassemble all readings or only those that are not "crucial" (those that would cause a contig to break into fragments). A non-zero value will disassemble all. The default is 1.

`-remove` *to_remove*
: Controls whether to completely remove the readings from the database or to move them to their own contigs. A non-zero value will remove them, otherwise they are moved. The default is 1.

---

double_strand

```
double_strand
      -io            io_handle:integer
      -contigs       identifiers:strings
     ?-max_nmismatch count:integer(8)
     ?-max_pmismatch percentage:float(8.)
```

This command searches for single stranded regions and attempts to make them double stranded data by finding neighbouring readings with hidden data that is good enough to reveal. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to double strand. The {*contig start end*} syntax may be used for an identifier to double strand only a region of the contig, otherwise all of it is double stranded.

`-max_nmismatch`This *cifies the maximum number of mismatches allowed in the extended dat*
between the reading and the consensus. The default is 8.

`-max_pmismatch`This *cifies the maximum percentage mismatch allowed in the extended dat*
between the reading and the consensus. The default is 8.0.

---

edit_contig

```
edit_contig
      -io           io_handle:integer
      -contig       identifier:string
     ?-reading      identifier:string()?
     ?-pos          position:integer(1)?
```

This command brings up a contig editor display. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
The database IO handle returned from a previous `open_db` call.

`-contig` *identifier*
This specifies the contig to edit.

`-reading` *identifier*
`-pos` *position*
By default the editor starts with the display and cursor at the left end of the consensus. Use these options to specify a different reading and position. The position is relative to the start of the specified reading. To start the editor at a specific position in the consensus sequence use only `-pos`.

---

enter_tags

```
enter_tags
      -io           io_handle:integer
      -file         filename:string
```

This command reads a file containing tags (annotations) and enters them into the database. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
The database IO handle returned from a previous `open_db` call.

`-file` *filename*
This specifies the file containing the tag data.

---

extract_readings

```
extract_readings
      -io           io_handle:integer
      -readings     identifiers:strings
     ?-directory    directory:string(extracts)?
     ?-quality      add_quality:integer(1)?
```

This command copies the edited sequences from the database to Experiment Files on disk. The database is not altered. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-readings` *identifiers*
: Specifies the list of readings to extract.

`-directory` *directory*
: The files created are all placed in a subdirectory (created by this command). This option specifies the directory to be used. The default is '`extracts`'.

`-quality` *add_quality*
: This controls whether quality, original positions, and pre-assembly format data is to be included in the file. A non-zero value will output the extra data. The default is 1.

---

find_long_gels

```
find_long_gels
      -io           io_handle:integer
      -contigs      identifiers:strings
     ?-avg_len      length:integer(500)?
```

This command searches for places where rerunning a reading as a long gel will solve a problem. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to search. The {*contig start end*} syntax may be used for an identifier to search only a region of the contig, otherwise all of it is searched.

`-avg_len` *length*
: This specifies the length expected for a long reading. This is used to determine which readings are suitable for rerunning and the amount of coverage available. The default is 500 base pairs.

---

find_oligo

```
find_oligo
      -io           io_handle:integer
      -contigs      identifiers:strings
     ?-min_pmatch   percentage:float(75.0)?
     ?-seq          sequence:string()?
     ?-tag_types    types:string()?
```

This command searches for short sequence segments and plots them in the Contig Selector. It will fail when not running in a graphical environment containing the Contig Selector. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to search. The {*contig start end*} syntax may be used for an identifier to search only a region of the contig, otherwise all of it is searched.

`-min_pmatch` *percentage*
: Only matches with this level of similarity or better will be displayed. The default is 75%.

`-seq` *sequence*
: The command will search for the *sequence* in each of the specified contigs, plotting matches above (or equal to) the mininum percentage match. This option takes precedence over the `-tag_types` option. The default is a blank string, which implies no searching.

`-tag_types` *types*
: If a list of tag types is specified the algorithm first obtains the sequence underneath each tag of these types. For each sequence the search is independently performed with that sequence as the search string. If `-seq` has also been specified this option is invalid. The default is a blank list of tag types, which implies no tags will be searched for.

---

find_primers

This command performs the Gap4 "Suggest Primers" function. It searches for locations where choosing an oligo primer for "walking" off another reading will solve a problem. The command returns a list of primer information in the form "*template_name reading_name primer_identifier sequence position direction*", separated by newlines.

```
find_primers
      -io             io_handle:integer
      -contigs        identifiers:strings()
     ?-search_from   position:integer(20)?
     ?-search_to     position:integer(60)?
     ?-num_primers   count:integer(1)?
     ?-primer_start  count:integer(1)?
     ?-params        OSP_params:string?
```

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to search. The {*contig start end*} syntax may be used for an identifier to search only a region of the contig, otherwise all of it is searched.

`-search_from` *position*
`-search_to` *position*
: These two options control the region, relative to the problem, in which to look for suitable oligos. The defaults are *from* 20 *to* 60, which means that to cover an area starting at position 1000 in the forward strand the command will pick oligos from the sequence at positions 940 to 980.

`-num_primers` *count*
: This controls how many oligos to pick to solve each problem. The default is 1.

`-primer_start` *count*
: Each oligo is given a primer name consisting of the database name followed by the primer number. The numbers start at *count* and increment for each new primer. The default is 1.

`-params` *OSP_params*
: This specifies the parameters to the OSP algorithm as a keyed list. The defaults are undefined unless the gaprc file has been parsed. In this case the defaults are as used by Gap4. Not all of the OSP parameters listed below are needed or used, but we don't have further details. The defaults listed in the gaprc file are:

```
#----------------------------------------------
# The OSP Prm defaults
#----------------------------------------------
set_def OSP.prod_len_low              0
set_def OSP.prod_len_high             200
set_def OSP.prod_gc_low               0.40
set_def OSP.prod_gc_high              0.55
set_def OSP.prod_tm_low               70.0
set_def OSP.prod_tm_high              90.0

set_def OSP.min_prim_len              17
set_def OSP.max_prim_len              23
set_def OSP.prim_gc_low               0.30
set_def OSP.prim_gc_high              0.70
set_def OSP.prim_tm_low               50
set_def OSP.prim_tm_high              55

set_def OSP.self3_hmlg_cut            8
set_def OSP.selfI_hmlg_cut            14
set_def OSP.pp3_hmlg_cut              8
set_def OSP.ppI_hmlg_cut              14
set_def OSP.primprod3_hmlg_cut        0
set_def OSP.primprodI_hmlg_cut        0
set_def OSP.primother3_hmlg_cut       0.0
set_def OSP.primotherI_hmlg_cut       0.0
set_def OSP.delta_tm_cut              2.0
set_def OSP.end_nucs                  S

set_def OSP.wt_prod_len               0
set_def OSP.wt_prod_gc                0
set_def OSP.wt_prod_tm                0
set_def OSP.wt_prim_s_len            0
set_def OSP.wt_prim_a_len            0
set_def OSP.wt_prim_s_gc             0
set_def OSP.wt_prim_a_gc             0
set_def OSP.wt_prim_s_tm             0
set_def OSP.wt_prim_a_tm             0
set_def OSP.wt_self3_hmlg_cut        2
set_def OSP.wt_selfI_hmlg_cut        1
set_def OSP.wt_pp3_hmlg_cut          2
set_def OSP.wt_ppI_hmlg_cut          1
set_def OSP.wt_primprod3_hmlg_cut    0
set_def OSP.wt_primprodI_hmlg_cut    0
set_def OSP.wt_primother3_hmlg_cut   0
set_def OSP.wt_primotherI_hmlg_cut   0
set_def OSP.wt_delta_tm_cut          0
set_def OSP.AT_score                 2
set_def OSP.CG_score                 4
set_def OSP.wt_ambig                 avg
```

To change a default you need to specify the full OSP parameters with modified values. For instance:

```
global gap_defs

set osp_defs [keylget gap_defs OSP]
keylset osp_defs min_prim_len 18

find_primers \
        -params $osp_defs \
        (etc)
```

find_probes

```
find_probes
      -io            io_handle:integer
      -contigs       identifiers:strings
     ?-min_size      length:integer(15)?
     ?-max_size      length:integer(19)?
     ?-max_pmatch    fraction:float(90.0)?
     ?-from          position:integer(10)?
     ?-to            position:integer(100)?
     ?-vectors       filename:string()?
```

This command performs the Gap4 "Suggest Probes" function. It searches for unique sequences at the ends of contigs suitable for probing clone libraries to pick overlapping sequences and hence to extend contigs. The command returns a newline separated list of probes in the form "`Contig` *ident* `position` *int* `Tm` *int* `sequence` *string*".

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to use. Only the contig identifier is currently used, although the syntax specifying start and end ranges is valid.

`-min_size` *length*
`-max_size` *length*
: These specify an inclusive range of the allowed lengths of probes chosen. The defaults are *min_size* of 15 and *max_size* of 19.

`-max_pmatch` *fraction*
: Each potential probe sequence is comparared against all contig sequences and, optionally, several vector sequences. This option specifies the maximum percentage match between the probe and the comparison sequences. Only sequences with no matches above this percentage match are considered unique. sequences. The default is 90.0.

`-from` *position*
`-to` *position*
: These specify the area in which to look for probes as an offset from the ends of the contigs. The defaults are *from* 10 *to* 100.

`-vectors` *filename*
: This specifies a file of vector filenames. NB: This will possibly be changed to a Tcl list of vector filenames. The uniqueness search will then also check the vector files for matches. The vector files can be in any format readable by the *seq_utils* library (currently Staden, EMBL, CODATA, GENBANK and FASTA). The default is blank, which implies no vectors to check.

---

find_read_pairs

```
find_read_pairs
      -io           io_handle:integer
      -contigs      identifiers:strings
```

This command searches for templates containing both forward and reverse readings where the forward and reverse readings are in differing contigs. This information is plotted on the Contig Selector. The command will not work if the Contig Selector is not displayed. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to use. Only the contig identifier is currently used, although the syntax specifying start and end ranges is valid.

---

find_repeats

```
find_repeats
      -io           io_handle:integer
      -contigs      identifiers:strings
     ?-direction    direction:integer(3)?
     ?-min_match    length:integer(25)?
     ?-outfile      filename:string()?
     ?-tag_types    types:string()?
```

The command searches for perfect matches between two or more fragments in the consensus sequences. This information is plotted on the Contig Selector. The command will not work if the Contig Selector is not displayed. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to search. The `{`*contig start end*`}` syntax may be used for an identifier to search only a region of the contig, otherwise all of it is searched.

`-direction` *direction*
: This specifies whether forward repeats (1), reverse repeats (2), or both (3) are to be found. The default is 3 (both).

`-min_match` *length*
: This specifies the minimum length of a repeat to be searched for. The default is 25. The minimum allowed is 8.

`-outfile` *filename*
: This specifies a file in which to save the tag hits. The results are written in the form of REPT annotations which are suitable for passing onto the `enter_tags` command. The default is a blank string, which implies no file should be created.

`-tag_types` *types*
: If *types* is a non blank list of tag types then masking will be applied to remove sequence covered by tags of these types from the repeat searching. The default is a blank list, which means no masking will be performed.

---

find_taq_terminator

```
find_taq_terminator
      -io           io_handle:integer
      -contigs      identifiers:strings
     ?-avg_len      length:integer(350)?
```

This command searches for places where rerunning a reading as a dye terminator reaction will solve a problem. Currently these places are identified by the presence of a COMP (compression) or STOP annotation. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to search. The `{`*contig start end*`}` syntax may be used for an identifier to search only a region of the contig, otherwise all of it is searched.

`-avg_len` *length*

This specifies the expected length achieved by a terminator reading . This is used to determine which readings are suitable for rerunning and for the amount of coverage available. The default is 350 base pairs.

---

find_internal_joins

```
find_internal_joins
      -io             io_handle:integer
      -contigs        identifiers:strings
     ?-mode           mode:string(end_all)?
     ?-segment        identifier:string()?
     ?-min_match      length:integer(15)?
     ?-max_pads       count:integer(25)?
     ?-max_pmismatch  percentage:float(30.0)?
     ?-win_size       length:integer(0)?
     ?-max_dashes     count:integer(0)?
     ?-probe_length   length:integer(100)?
     ?-mask           mask:string(none)?
     ?-tag_types      types:string()?
```

This command searches for potential joins between contigs by comparing the sequence data in each contig. This information is plotted on the Contig Selector. The command will not work if the Contig Selector is not displayed. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*

The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*

This specifies the list of contigs to search. The {*contig start end*} syntax may be used for an identifier to search only a region of the contig, otherwise all of it is searched.

`-mode` *mode*

This specifies the segments of contigs in which to search for joins. Valid *modes* are:

`end_end`	Compares only the ends of each contigs with the ends of other contigs.
`end_all`	Compares the ends of each contigs with the entirety of other contigs.
`all_all`	Compares all of each contig with all of the other contigs.
`segment`	Compares a segment of a particular contig with all of the contig contigs.

The default mode is `"end_all"`.

`-segment` *identifier*

When *mode* is `"segment"` this specifies the region of the contig identifier to compare. The default is blank.

`-min_match` *length*

Specifies the minimum length of exact match used during the hashing stage of find internal joins. The minium allowed value for this is 14. The default is 15.

`-max_pads` *count*
: After alignments the number of pads required in each of the two consensus sequences must be less than or equal to *count*. The default is 25.

`-max_pmismatch` *count*
: After alignments the percentage of bases that do not match must be less than or equal to *percentage*. This is a floating point value. The default is 30.0.

`-win_size` *length*
`-max_dashes` *count*
: If these are both set to non-zero values the cutoff data will be searched for matches. The criteria for determining how much cutoff sequence to align is selected as the portion where no more than *count* unknown (`"-"`) bases within a region of *length* bases. The defaults are 0 for both, which implies only used data should be searched.

`-mask` *mask*
`-tag_types` *mask*
: If *types* is a non blank list of tag types then masking or marking will be applied to the sequence covered by tags of these types from. When *mask* is `"mark"` the sequence is converted to an alternative character set so that matches will be found, but are clearly visible in the output as being matches between marked fragments. When *mask* is `"mask"` the sequence is removed so that no matches will be initiated between this sequence and another fragment. The defaults are`" none"` for *mask* and a blank string for the tag types, which disables masking and marking.

---

get_consensus

```
get_consensus
      -io           io_handle
      -contigs      identifiers:strings
      -outfile      filename:string
     ?-type         type:string(normal)?
     ?-mask         mask:string(none)?
     ?-tag_types    types:string()?
     ?-win_size     length:integer(0)?
     ?-max_dashes   count:integer(0)?
     ?-format       format:integer(3)?
     ?-annotations  annotations:integer(0)?
     ?-truncate     truncate:integer(0)?
```

This command calculates the consensus sequence for one or more contigs and saves it to a file. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to search. The {*contig start end*} syntax may be used for an identifier to search only a region of the contig, otherwise all of it is searched.

`-outfile` *filename*
: Specifies the filename to write the consensus sequence too. This has no default value.

`-type` *type*
: This specifies the final output type for the consensus algorithm. Valid *type*s are:

  `normal` The standard consensus sequence consisting of A, C, G, T, - and *.

  `extended` As per `normal`, except the cutoff data at the ends of contigs is used to provide consensus sequence beyond the well defined contig ends.

  `unfinished`
  The consensus sequence in single stranded regions is output as a, c, g and t whilst the consensus for finished regions is listed as d, e f and i (for a, c, g and t respectively). The quality of each base is output instead of the consensus base. The base quality is listed as a single letter from the following table showing the quality of each strand independently.

a	*Good Good (in agreement)*
b	*Good Bad*
c	*Bad Good*
d	*Good None*
e	*None Good*
f	*Bad Bad*
g	*Bad None*
h	*None Bad*
i	*Good Good (disagree)*
j	*None None*

`-win_size` *length*
`-max_dashes` *count*
: These are only of use during the *extended* consensus type. The criteria for determining how much cutoff sequence to output is selected as the portion where there are no more than *count* unknown (`"-"`) bases are found within a region of *length* bases. The defaults are 0 for both, which implies that only used data should be output.

`-format` *format*
: Specifies the output format of the file to be created. All formats can be written for all consensus types, but some may not be legal (eg Fasta files containing quality codes instead of sequence). The available formats are:

`1`	Staden format
`2`	Fasta format
`3`	Experiment File format

  The default is 3.

`-annotations` *annotations*
: This controls whether to output annotations. This is only of used in the Experiment File output format. Note that with the *extended* consensus type the annotation positions are still for the *normal* consensus; this is a bug which will only be fixed if it is considered useful. A non-zero value will output annotations. The default is 0, which is to not output annotations.

`-truncate` *truncate*
: This controls whether annotations within or overlapping the cutoff data will be output. A non-zero value will not output annotations within the cutoff data. The default is 0.

`-mask` *mask*
`-tag_types` *types*
: If *types* is a non blank list of tag types then masking or marking will be applied to the sequence covered by tags of these types from. When *mask* is `"mask"` the sequence is converted to an alternative character set (*d*, *e*, *f* and *i* for Experiment Files and Staden format and *n*s for Fasta format). When *mask* is `"mark"` the sequence is in lowercase. The defaults are" `none`" for *mask* and a blank string for the tag types, which disables masking and marking. Masking and marking is only used in the *normal* and *extended* consensus types.

---

join_contig

```
join_contig
      -io          io_handle:integer
      -contig1     identifier:string
     ?-reading1    identifier:string()?
     ?-pos1        position:integer(1)?
      -contig2     identifier:string
     ?-reading2    identifier:string()?
     ?-pos2        position:integer(1)?
```

This command brings up a join editor display. The display consists of two contig editors, one above the other. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contig1` *identifier*
`-contig2` *identifier*
: These specify the contigs to join.

`-reading1` *identifier*
`-reading2` *identifier*
`-pos1` *position*
`-pos2` *position*
: By default the editors start with the display and cursor at the left end of the consensus. Use these options to specify a different reading and position. The position is relative to the start of the specified reading. To start the editors at specific positions in the consensus sequence use only `-pos`*n*.

---

minimal_coverage

```
minimal_coverage
      -io           io_handle:integer
      -contigs      indentifiers:strings
```

This command produces a list of readings that, between them, cover the full length of the contig. The algorithm may not produce the optimum set of readings, but the result is at least close to optimum. The command returns the minimal list of readings.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: This specifies the list of contigs to use. Only the contig identifier is currently used, although the syntax specifying start and end ranges is valid.

---

pre_assemble

```
pre_assemble
      -io           io_handle:integer
      -files        filenames:strings
```

This command performs the Gap4 "Enter Preassembled Data" function. It assembles data that contains the PC, SE, ON and AV Experiment File line types to specify exactly the position data. This is superseded by the assemble_direct command and should no longer be used. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-files` *filenames*
: *Filenames* must contain a Tcl list of files to assemble.

---

shift_readings

```
shift_readings
      -io           io_handle:integer
      -readings     identifiers:integers
      -distances    distances:integers
```

This command shifts all readings to the right of (and including) a specified reading by a particular amount to the left or right. It is mainly for manually manipulating the database structures to join contigs. Use is not recommended. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-readings` *readings*
`-distances` *distances*
: For each reading listed in the *readings* argument, that reading and all those to the right of it are shifted by the corresponding element in the *distances* list. Positive distances shift right; negative distances shift left.

---

show_relationships

```
show_relationships
      -io            io_handle:integer
     ?-contigs       identifiers:strings()?
     ?-order         order:integer(1)?
```

This command performs the Gap4 Show Relationships function. The function returns no value but will generate a Tcl error if an error occurs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

`-contigs` *identifiers*
: Specifies single segments of contigs for which to display the relationships data. In the current implementation only the first contig (and start and end position) in the identifier list is processed. Not specifying any contigs (which is the default) will make show_relationships process all contigs.

`-order` *order*
: Controls whether the output should be sorted on positional order or reading number order. This has no effect when `-contigs` is used. An *order* of 1 specifies that the output will list each contig in turn together with the readings within that contig listed in positional order. An *order* of 0 lists all contig records first followed by all readings in contig and reading number order. The default is 1.

---

unattached_readings

```
unattached_readings
      -io            io_handle:integer
```

This command produces a list of the contigs which are single readings. The command returns a Tcl list of the reading identifiers for these contigs.

`-io` *io_handle*
: The database IO handle returned from a previous `open_db` call.

2.5 The Editor Widget

2.5.1 Introduction

2.5.2 Configuration Options

These options are specified when creating the editor widget to configure its look and feel. In addition to the options listed below the editor supports the `-width`, `-height`, `-font`, `-borderWidth`, `-relief`, `-foreground`, `-background`, `-xscrollcommand` and `-yscrollcommand`. These are described in detail in the Tk *options* manual page. Note that the `-width` and `-height` values are measured in characters.

In the descriptions below 'Command-Line Name' refers to he switch used in class commands and `configure` widget commands to set this value. 'Database Name' refers to the option's name in the option database (e.g. in '`.Xdefaults`' files). 'Database Class' refers to the option's class value in the option database.

Command-Line Name: `-lightcolour`
Database Name: `lightColour`
Database Class: `Foreground`

> Specifies the foreground colour to use when displaying the cutoff data.

Command-Line Name: `-max_height`
Database Name: `maxHeight`
Database Class: `MaxHeight`

> Specifies the maximum height the editor is allowed to display, in units of characters. The vertical scrollbar will be used when more than this many sequences are displayed.

Command-Line Name: `-qualcolour`*n* ($0 <= n <= 9$)
Database Name: `qualColour`*n*
Database Class: `Background`

> These specify the 10 colours to be used for the background of the bases when `show_quality` is enabled. `-qualcolour0` should be the darkest (defaults to '`#494949`') and `-qualcolour9` should be the lightest (defaults to '`#d9d9d9`').

Command-Line Name: `-qual_fg`
Database Name: `qualForeground`
Database Class: `Foreground`

> This specifies the foreground colour of bases with poorer quality than the current quality cutoff. By default this is redish ('`#ff5050`').

2.5.3 Widget Commands

The 'editor' widget is based upon the sheet display widget except with a large range of editing commands added. The data for the editor cannot be specified from the Tcl level, rather this requires using a C interface to adjust the tkEditor structure. Hence the editor widget is very specific for the task at hand.

2.5.3.1 Units and Coordinates

The contig editor works in base coordinates. Some widget commands take x and/or y position arguments. These are by default in base units. However it is possible to use '`@pos`' as the position argument to specify '`pos`' as pixel units.

2.5.3.2 The Editing Cursor

`cursor_left`
`cursor_right`
`cursor_up`
`cursor_down`
: Move the editing cursor in the appropriate direction. The exact allowed movements depends on where the cursor is and whether cutoff data is displayed.

`read_start`
`read_end`
: Positions the cursor at the beginning or end of the used data for this sequence.

`read_start2`
`read_end2`
: Positions the cursor at the beginning or end of the displayed data for this sequence. These differ from `read_start` and `read_end` when cutoff data is displayed in that they use the ends of the cutoff data.

`contig_start`
`contig_end`
: Positions the cursor on the consensus line at the start or end of the contig.

`cursor_set` *xpos ypos*
: Positions the cursor at the correct position and sequence based on an (x,y) coordinate pair from the topleft corner of the screen. Units are in bases unless '`@`*xpos* `@`*ypos*' is used, in which case they are pixels.

`cursor_consensus ?`*xpos*`?`
: Positions the cursor at an absolute position within the consensus. If no *xpos* is given the existing position within the contig is returned.

2.5.3.3 The Selection

The widget supports the standard X selection via the '`select`' command. The general form of this command is '`select` *option ?arg?*'. A selection here is simply a portion of text. Selections can be made on any sequence or consensus sequence and are denoted by being underlined.

`select clear`
: Clears and disowns the current selection.

`select from` *pos*
: Grabs the current selection and sets it's start position.

`select to` *pos*
`select adjust` *pos*
: Currently both these are the same. They set the end position of the selection.

2.5.3.4 Cutoff Adjustments

The consensus calculation can be tuned by changing the threshhold at which a particular base type is considered to have the 'majority'; a dash (-) is displayed when the majority is not sufficiently high. See the staden package manual for precise details on this.

An additiona quality cutoff can be applied to each base. This determines the contribution that each base makes to the consensus calculation and also the colour used when displaying bases on the screen. Bases with a quality lower than the cutoff are displayed in `qualColour` and `qualForeground` colours, as defined in the configuration Options listed above.

`set_ccutoff ?`*value*`?`
: If *value* is specified the consensus cutoff is set to *value*. Otherwise the existing consensus cutoff value is returned without making any changes.

`set_qcutoff ?`*value*`?`
: If *value* is specified the quality cutoff is set to *value*. Otherwise the existing quality cutoff value is returned without making any changes.

2.5.3.5 Annotations

`delete_anno`
: Delete the tag underneath the cursor. This also sets the current selection to be the range covered by the tag.

`create_anno`
: Brings up a tag editor window to create a new tag. This requires the selection to have been previously set.

`edit_anno`
: Brings up a tag editor window. This also sets the current selection to be the range covered by the tag.

2.5.3.6 Editing Commands

`transpose_left`
`transpose_right`
: Moves a base in a sequence either left or right one character. Does not work on the consensus sequence. Only pads can be moved unless the appropriate superedit mode is enabled.

`extend_left`
`extend_right`
`zap_left`
`zap_right`
: Adjusts the current left or right cutoff for a sequence. The `extend_` commands move the cutoff by a single base and require the editing cursor to be at the appropriate end of the used data. The `zap_` commands set the appropriate end of the used data to be the current cursor position.

`delete_key`
`delete_left_key`
: Delete comes in two modes. Both delete the base to the left of the editing cursor. `delete_key` then moves the sequence to the right and the editing cursor left by one base to fill the removed base. `delete_left_key` moves the sequence to the left of the editing cursor right by one base, and hence changes the sequence start position too. Typically the *`DEL`* key is bound to `delete_key` and *`CTRL-DEL`* is bound to `delete_left_key`.

`edit_key` *character*
: Other general key presses. Typically any other key press is bound to this call, which then handles the actual editing or replacing of bases. The key character should be passed over as an argument.

`set_confidence` *value*
: Sets the confidence value of a base to *value*. In the current implementation only values of 0 and 100 are allowed.

2.5.3.7 Editing Toggles and Settings

The editor has a variety of boolean values for determining editing and display modes. Most take an optional *value* parameter to explicitly set the value of the boolean. With no *value* parameter specified the boolean is toggled instead.

`set_reveal` ?*value*?
: This sets the editor 'cutoffs' mode. A setting of 1 indicates that cutoff data is to be displayed in the `lightColour` colour. A setting of 0 indicates that no cutoff data is to be displayed.

`set_insert` ?*value*?
: This command sets the editor insert/replace mode. A *value* of 1 sets the editor to insert mode. A *value* of 0 sets the editor to replace mode.

`superedit` *modes*
: This command sets which editing actions should be allowed. The *modes* argument should be a Tcl list of 10 values, each 0 (disabled) or 1 (enabled). The values in order repesent insert any to read, delete any from read, insert to consensus, delete dash from consensus, delete any from consensus, replace base in consensus, shift readings, transpose any bases, can use uppercase edits, and replacement mode. The replacement mode is 0 for editing by base type and 1 for edit by confidence value.

`auto_save` ?*value*?
: This command sets the auto-save mode. A *value* of 1 enables auto-saving. A *value* of 0 disables it.

`show_differences` ?*value*?
: This command set the show differences mode. A *value* of 1 will display only those bases that disagree with the consensus. All other bases are displayed as a fullstop. A *value* of 0 shows all bases.

`compare_strands` ?*value*?
: This command sets the compare strands mode. A *value* of 1 will make the editor compute the consensus separately for the positive and negative strands. Strands that disagree are given a final consensus character of '-'. A *value* of 0 will use the normal single consensus mode.

`join_lock` ?*value*?
: This command sets the scroll locking between two editors forming a join editor. A *value* of 1 will mean that scrolling (not cursor movement) in one contig will also scroll the other contig.

`show_quality` *value*
: This commands sets the quality display mode. With a *value* of 1 and a positive quality cutoff value all qualities values are displayed as grey scales using the 10 `qualColour`*n* configuration options.

2.5.3.8 Searching

The editor search procedures search for a particular item and move the editing cursor and xview position if a search item is found. Each search command takes a direction and a search string. *Direction* can be either '`forward`' or '`reverse`'.

`search` *direction* `name` *value*
: Searches for the reading name starting with *value*.

`search` *direction* `anno` ?*value*?
: Searches for the annotation containing a comment matching the *value* regular expression. Not specifying *value* will match all annotations.

`search` *direction* `sequence` *value*
: Searches for the sequence *value* using a case-insensitive exact match.

`search` *direction* `tag` *value*
: Searches for a tag with type *value*.

`search` *direction* `position` *value*
: Moves to a specific position. If *value* is an absolute number (eg '`30717`' then the editing cursor is moved to that consensus base. If *value* is '`@`' followed by a number (eg '`@100`') then the editing cursor is moved to that base within the current reading. If *value* starts with a plus or minus the editing cursor is moved forwards or backwards by that amount. The *direction* parameter here has no effect and is included purely for consistency.

`search` *direction* `problem`
: Searches for undefined bases or pads.

`search` *direction* `quality`
: Searches for bases of poor quality (undefined bases, pads, or single stranded data).

`search` *direction* `edit`
: Searches for sequence edits, including confidence value changes.

`search` *direction* `verifyand`
`search` *direction* `verifyor`
: Searches for consensus bases that have a lack of evidence in the original sequences. `verifyand` looks for evidence on both strands together. `verifyor` looks for evidence on each strand independently and defines a match to be places where either strand has a lack of evidence. In the current implementation of these two searches only the forward direction is supported.

2.5.3.9 Primer Selection

These control the searching for and creation of oligo primers. Together they form the Select Primer functionality of the contig editor. The `generate` command must be run first. All other commands have undefined behaviour when the generate command has not been run since the last quit command.

`select_oligos generate` *sense forward backward avg_length*
: Generates a list of oligos suitable for use on the *sense* strand, within *forward* bases rightwards of the cursor and *backward* bases leftwards. Returns the number of oligos found, or -1 for error.

`select_oligos next`
: Picks the next oligo in the list produced by the `generate` command (or the first if this hasn't been called yet). This remembers the current active oligo number and returns the default template followed by the complete list of templates (including the default) suitable for this oligo.

`select_oligos accept` *template*
: Adds the tag to the database for this oligo using the named *template* (`""` can be specified here if none is required). Returns a status line containing the template name and the oligo sequence.

`select_oligos quit`
: Frees up memory allocated by `generate` command.

2.5.3.10 The Status Line

`status add` *type*

Adds a new status line to the bottom of the editor. The *type* may be one of the following.

0	Strand display
1, 2 or 3	Amino acid translations in reading frame 1, 2 and 3 for the positive strand.
4, 5 or 6	Amino acid translations in reading frame 1, 2 and 3 for the negative strand.

status `delete` *type*

Delete a status line. The *type* is from the same set listed above.

`translation_mode` *mode*

This command sets the style of amino acids displayed. *Mode* may be either `1` or `3` to output 1 character or 3 character abbreviations.

2.5.3.11 The Trace Display

`autodisplay_traces ?`*value*`?`

This command sets the automatic trace display mode. A *value* of 1 will make the editor display relevant traces to solve a problem when the `problem` search type is used. A *value* of 0 disables this.

`set_trace_lock ?`*value*`?`

This command sets the locking mode between the editor cursor and the trace cursor. With a *value* of 1 any movement in the editor cursor also moves the connected trace displays. A *value* of 0 disables this.

`trace_comparator ?`*identifier*`?`

This command specifies another reading identifier (within the same contig) to compare all new traces against. The comparator *identifier* can either be a reading identifier to compare against that specific reading or `0` to compare against a consensus trace. When `invoke_trace` is called the comparator trace, the requested trace, and their differences are displayed. With no *identifier* argument the automatic trace comparison is disabled.

`trace_config ?`*match select*`?`

This command controls of generation of the consensus trace when `trace_comparator 0` is used. The consensus trace is calculated as the average trace of readings on the same strand as the trace we wish to compare it against. If *match* is non zero, each single base segment of the consensus trace is averaged from only readings in agreement with the consensus sequence. If *select* is non zero the trace to compare against is not used in the consensus trace calculation. With no *match* or *select* arguments the current settings are returned.

`delete_trace` *path*

Removes a trace with the Tk *path* from the trace display. Useful for when quitting the editor.

`invoke_trace`

Adds a trace to the trace display.

`diff_trace` *path1 path2*

This brings up a difference trace between two currently displayed traces with the Tk pathnames of *path1* and *path2*. These pathnames are returned from the initial `trace_add` and `trace_create` Tcl utility routines and are typically only known internally to the editor.

2.5.3.12 Miscellaneous Commands

`xview` ?*position*?
`yview` ?*position*?
: These commands are used to query and change the horizontal and vertical position of the information displayed in the editor's window. Without specifying the optional *position* argument the current value is returned. Specifying *position* sets the position and updates the editor display.

`align`
: Aligns the data covered by the selection with the consensus sequence. The sequence is then padded automatically.

`configure` ?*option*? ?*value option value ...*?
: Reconfigures the editor. NB: not all configuration options allowed at startup operate correctly when reconfiguring. (This is a bug.)

`dump_contig` *filename from to line_length*
: Saves the contig display to a file within a specified region. The output consists of the data and settings of the current display.

`edits_made`
: Queries whether edits have been made. Returns 1 if they have, 0 if they have not.

`find_read` *identifier*
: Converts a reading identifier to an internal editor sequence number.

`get_displayed_annos`
: Returns a list of the displayed annotation types.

`get_extents`
: Returns the start and end of the displayable contig positions. If cutoff data is shown this will also include the cutoff data beyond the normal contig ends.

`get_hidden_reads`
: This returns the hidden reads as a list of reading name identifiers.

`get_name` ?*gel_number*?
: Returns the gel name from a given internal reading number, or for the reading underneath the editing cursor.

`get_number` ?*xpos ypos*?
: Returns the editor's internal reading number covering the screen coordinate (*xpos*,*ypos*). If no *xpos* and *ypos* are specified then the position of the editing cursor is used.

`get_read_number` ?*xpos ypos*?
: Returns the reading number covering the screen coordinate (*xpos*,*ypos*). If no *xpos* and *ypos* are specified then the position of the editing cursor is used.

`hide_read`
: This command toggles the 'hidden' status of a reading. Hidden readings are not used to compute the consensus.

`io`
: This returns the IO handle used for this editor.

`join`
: Performs a join in the join editor.

`join_align`
: Performs an alignment (and pads automatically) on the overlapping region in a join editor.

`join_mode`
: Queries whether the editor is part of a join editor. Returns 1 if it is and 0 if it is not.

`join_percentage`
: Returns the percentage mismatch of the overlap for a join editor.

`save`
: Saves the database, but doesn't quit.

`set_displayed_annos ?`*type ...*`?`
: Sets the displayed annotation types to those specified. All other are turned off.

`shuffle_pads`
: Realigns pads along the total length of the consensus.

`undo`
: Undoes the last compound operation (from a list of changes).

`write_mode`
: This toggles the editor between read-write and read-only mode.

2.5.3.13 Quitting the Widget

Destroying the editor widget automatically destroys the associated data (edStruct) and deregisters from the contig. However a quit command also exists. The difference between using the Tk destroy command and quit is that quit also sends acknowledgements of shutdown allowing other displays to tidy up (such as deleting a displayed cursor). Hence quit is the preferred method.

`quit`
: Destroys the widget.

2.6 The EdNames Widget

This widget is currently undocumented. See the '`src/gap4/tkEdNames.c`' file.

3 Database I/O in C

3.1 Introduction and Overview

[General notes to go somewhere: It is better to check success return codes rather than failure ones as the failure ones are often variable (-1, 1, >0, etc) but most return 0 for success.]

The Gap4 I/O access from within C consists of several layers. These layers provide ways of breaking down the tasks into discrete methods, and of hiding most of the implementation details. For the programmer willing to extend Gap4, only the higher layer levels are of interest. Hence the lowest levels are described only briefly.

3.1.1 "g" Level - Raw Database Access

At the final end of any I/O is the actual code to read and write information to the disk. In Gap4 this is handled through a library named `"g"`. This contains code for reading, writing, locking and updating of the physical database. It does not describe the structures contained in the gap database format itself, but rather provides functions to read and write arbitrary blocks of data. Don't delve into this unless you're feeling brave!

The code for this library is contained within the '`src/g`' directory. No documentation is currently available on these functions.

3.1.2 "Communication" Level - Interfaces to the "g" Level

This level of code deals with describing the real Gap4 data structures and the interfacing with the g library. Generally this code should not be used.

This code is contained within the '`src/gap4`' directory and breaks down as follows:

'`gap-if.c`'
'`gap-local.c`'
'`gap-remote.c`'
: Interface functions with the g library. These are to provide support for a local (ie compiled in) or remote (unimplemented) database server.

'`gap-io.c`'
: Contains `GAP_READ` and `GAP_WRITE` functions in byte swap and non byte swap forms (depending on the system arch.). The `gap_io_init()` function automatically determines the machine endian and sets up function pointers to call the correct functions.

'`gap-error.c`'
: Definitions of `GAP_ERROR` and `GAP_ERROR_FATAL` functions.

'`gap-dbstruct.c`'
'`gap-create.c`'
: Functions for creation, initialisation, and copying of database files.

'`gap-dbstruct.h`'
: **VERY USEFUL!** The definitions of the gap structures that are stored in the database.

'`gap-init.c`'
: Initialises communication with the "g" database server by use of `gap_init()`, `gap_open_server()` and `gap_shutdown_server()` functions.

No documentation is currently available on these functions.

3.1.3 Basic Gap4 I/O

This level contains the basic functions for reading, writing, creation and deletion of the Gap4 structures, such as readings and templates as well as higher level functions built on top of these. It is this level of code that should generally be used by the programmer. The implementation of this level has function code and prototypes spread over a variety of files, but the programmer should only `#include` the '`IO.h`' file.

The primary functions are:

'`IO.c`'

`open_db`
`close_db`
`del_db` Opening/creation, closing and deletion of databases.

`GT_Read, GT_Write, GT_Write_cached`
`TextRead, TextAllocRead, TextWrite`
`DataRead, DataWrite`
`ArrayRead, ArrayWrite`
`BitmapRead, BitmapWrite`
: The basic IO calls. Note that the GT ones are for handling structures (eg GReadings) and the others for data of the associated type.

`io_init_contig`
`io_init_annotations`
`io_init_reading`
: Some functions for initialising new data structures. These in turn call the `allocate()` function to create new database records.

`io_read_seq`
`io_write_seq`
: Reads and writes sequence information.

`io_read_rd`
: Fetches the trace type and name values for a reading.

`io_read_annotation`
`io_write_annotation`
: Reading and writing of annotations (also known as tags).

`allocate`
`deallocate`
`io_deallocate_reading`
: Allocation and deallocation of records.

`flush2t` Flushes changes back to disk. The various write commands write the data to disk, but until a flush occurs they will not be committed as the up to date copies.

'`io_handle.c`'

`io_handle`
`handle_io`
: Converts between C *GapIO* pointer and an integer value which can be passed around in Tcl and Fortran. The integer handle is used in the Tcl scripting language.

'`io_utils.[ch]`'

`get_gel_num, lget_gel_num`
`get_contig_num, lget_contig_num`
: Converts single or lists of reading identifiers into reading or contig numbers (with start and end ranges).

`to_contigs_only`
: Converts a list of reading identifiers to contig numbers.

`get_read_name`
`get_contig_name`
`get_vector_name`
`get_template_name`
`get_clone_name`
: Converts a structure number into its textual name.

`chain_left`
: Finds the left most reading number in a contig from a given reading number.

`rnumtocnum`
: Converts from a reading number into a contig number.

3.1.4 Other I/O Functions

Still more I/O functions exist that aren't listed under the "Basic Gap4 I/O" header. The reason for this is primarily due to code structure rather than any particular grouping based on functionality. Specifically, these functions cannot be easily linked into "external" applications without a considerable amount of effort.

The file break down is as follows.

'`IO2.c`'

`io_complement_seq`
: Complements, in memory, a sequence and associated structures.

`io_insert_seq`
`io_delete_seq`
`io_replace_seq`
: Modifies in memory sequence details.

`io_insert_base`
`io_modify_base`
`io_delete_base`
: Modifies a single base in a sequence on the disk.

`pad_consensus`
: Inserts pads to the consensus sequence and all the readings at that point.

`io_delete_contig`
: Removes a contig structure.

'`IO3.c`'

`get_read_info`
`get_vector_info`
`get_clone_info`
: Fetches miscellaneous information for reads (primers, insert size, etc), vectors and clones.

`io_get_extension`
: Returns the right cutoff of a reading. Found by checking the cut points and any vector tags.

`io_mod_extension`
: Modifies the cutoffs of readings.

`write_rname`
: Updates a reading name in memory and disk.

3.2 Compiling and Linking with Other Programs

If you require usage of the Gap4 I/O functions in a program other than Gap4 itself you will need to compile and link in particular ways to use the function prototypes and to add the Gap4 functions to your binary. At present, the object files required for database access do not comprise a library.

The compiler include search path needs adjusting to add the '`$STADENROOT/src/gap4`' directory and possibly the '`$STADENROOT/src/g`' directory. Once your own object files are compiled, they need to be linked with the following gap4 object files.

```
$STADENROOT/src/gap4/$MACHINE-binaries/actf.o
$STADENROOT/src/gap4/$MACHINE-binaries/gap-create.o
$STADENROOT/src/gap4/$MACHINE-binaries/gap-dbstruct.o
$STADENROOT/src/gap4/$MACHINE-binaries/gap-error.o
$STADENROOT/src/gap4/$MACHINE-binaries/gap-if.o
$STADENROOT/src/gap4/$MACHINE-binaries/gap-init.o
$STADENROOT/src/gap4/$MACHINE-binaries/gap-io.o
$STADENROOT/src/gap4/$MACHINE-binaries/gap-local.o
$STADENROOT/src/gap4/$MACHINE-binaries/gap-remote.o
$STADENROOT/src/gap4/$MACHINE-binaries/IO.o
$STADENROOT/src/gap4/$MACHINE-binaries/io_handle.o
$STADENROOT/src/gap4/$MACHINE-binaries/io-reg.o
$STADENROOT/src/gap4/$MACHINE-binaries/io_utils.o
$STADENROOT/src/gap4/$MACHINE-binaries/text-io-reg.o
```

Finally, a library search path of '`$STADENROOT/lib/$MACHINE-binaries`' should be used to link the `-lg -ltext_utils -lmisc` libraries.

All of the above definitions have been added to a single Makefile held in '`$STADENROOT/src/mk/gap4_defs.mk`' as the `GAPDB_EXT_INC`, `GAPDB_EXT_OBJS` and `GAPDB_EXT_LIBS` variables. When possible, these should be used in preference to hard coding the variable object filenames as this provides protection against future coding changes. So for example, if we have a program held in the file '`demo.c`' we could have a simple Makefile as follows.

```
SRCROOT=$(STADENROOT)/src
include $(SRCROOT)/mk/global.mk
include $(SRCROOT)/mk/$(MACHINE).mk

OBJS = $(O)/demo.o

LIBS = $(MISC_LIB)

$(O)/demo: $(OBJS)
        $(CLD) -o $ $(OBJS) $(LIBS) $(LIBSC)
```

If we now extend this program so that it requires the Gap4 I/O routines, the Makefile should be modified to:

```
SRCROOT=$(STADENROOT)/src
include $(SRCROOT)/mk/global.mk
include $(SRCROOT)/mk/$(MACHINE).mk
include $(SRCROOT)/mk/gap4_defs.mk

INCLUDES_E += $(GAPDB_EXT_INC)

OBJS = $(O)/demo.o $(GAPDB_EXT_OBJS)

LIBS = $(MISC_LIB) $(GAPDB_EXT_LIBS)

$(O)/demo: $(OBJS)
        $(CLD) -o $ $(OBJS) $(LIBS) $(LIBSC)
```

If you require an example of a program that utilises the Gap4 I/O functions, see the `convert` program in '`$STADENROOT/src/convert/`'.

3.3 Database Structures

Before using any of the functions a firm understanding of the data structures is needed. The main objects held within the database are readings, contigs, templates, vectors, clones and annotations. These reference additional records of other objects or one of the primitive types.

There are five basic types from which the database structures are constructed. These are:

GCardinal	A single 4 byte integer.
Text	An ascii string which may ending in a null. The null character may, or may not, be present in the actual data stored on the disk.
Array	An extendable list of 4 byte integer values.
Bitmap	An extendable array of single bits.
Data	Any other data. This is handled in a similar manner to the Text type except the null character may be present.

In the C code, the *GCardinal* is the basic type used in most database structures. Other structure elements are larger and so are typically stored as another `GCardinal` containing the record number of the data itself.

3.3.1 The GDatabase Structure

```
#define GAP_DB_VERSION 2
#define GAP_DNA    0
#define GAP_PROTEIN    1

typedef struct {
    GCardinal version; /* Database version - GAP_DB_VERSION */
    GCardinal maximum_db_size; /* MAXDB */
    GCardinal actual_db_size; /* */
    GCardinal max_gel_len; /* 4096 */
    GCardinal data_class; /* GAP_DNA or GAP_PROTEIN */

    /* Used counts */
```

```
        GCardinal num_contigs; /* number of contigs used */
        GCardinal num_readings; /* number of readings used */

        /* Bitmaps */
        GCardinal Nfreerecs; /* number of bits */
        GCardinal freerecs; /* record no. of freerecs bitmap */

        /* Arrays */
        GCardinal Ncontigs; /* elements in array */
        GCardinal contigs; /* record no. of array of type GContigs */

        GCardinal Nreadings; /* elements in array */
        GCardinal readings; /* record no. of array of type GReading */

        GCardinal Nannotations; /* elements in array */
        GCardinal annotations; /* record no. of array of type GAnnotation */
        GCardinal free_annotations; /* head of list of free annotations */

        GCardinal Ntemplates; /* elements in array */
        GCardinal templates; /* record no. of array of type GTemplates */

        GCardinal Nclones; /* elements in array */
        GCardinal clones; /* record no. of array of type GClones */

        GCardinal Nvectors; /* elements in array */
        GCardinal vectors; /* record no. of array of type GVectors */

        GCardinal contig_order; /* record no. of array of type GCardinal */

        GCardinal Nnotes; /* elements in array */
        GCardinal notes_a; /* records that are GT_Notes */
        GCardinal notes; /* Unpositional annotations */
        GCardinal free_notes; /* SINGLY linked list of free notes */
    } GDatabase;
```

This is always the first record in the database. In contains information about the Gap4 database as a whole and can be viewed as the root from which all other records are eventually referenced from. Care must be taken when dealing with counts of contigs and readings as there are two copies; one for the used number and one for the allocated number.

The structure contains several database record numbers of arrays. These arrays in turn contain record numbers of structures. Most other structures, and indeed functions within Gap4, then reference structure numbers (eg a reading number) and not their record numbers. The conversion from one to the other is done by accessing the arrays listed in the GDatabase structure.

For instance, to read the structure for contig number 5 we could do the following.

```
GContigs c;
GT_Read(io, arr(GCardinal, io->contigs, 5-1), &c, sizeof(c), GT_Contigs);
```

In the above code, `io->contigs` is the array of GCardinals whose record number is contained within the *contigs* element of the GDatabase structure. In practise, this is hidden away by simply calling `"contig_read(io, 5, c)"` instead.

version Database record format version control. The current version is held within the `GAP_DB_VERSION` macro.

maximum_db_size
actual_db_size
These are essentially redundant as Gap4 can support any number of readings up to *maximum_db_size*, and *maximum_db_size* can be anything the user desires. It is specifable using the `-maxdb` command line argument to gap4.

max_gel_len
This is currently hard coded as 4096 (but is relatively easy to change).

data_class This specifies whether the database contains DNA or protein sequences. In the current implementation only DNA is supported.

num_contigs
num_readings
These specify the number of **used** contigs and readings. They may be different from the number of records allocated.

Nfreerecs
freerecs *freerecs* is the record number of a bitmap with a single element per record in the database. Each free bit in the bitmap corresponds to a free record. The *Nfreerecs* variable holds the number of bits allocated in the freerecs bitmap.

Ncontigs
contigs *contigs* is the record number of an array of GCardinals. Each element of the array is the record number of a GContigs structures. *Ncontigs* is the number of elements allocated in the *contigs* array. Note that this is different from *num_contigs*, which is the number of elements used.

Nreadings
readings *readings* is the record number of an array of GCardinals. Each element of the array is the record number of a GReadings structures. *Nreadings* is the number of elements allocated in the *readings* array. Note that this is different from *num_readings*, which is the number of elements used.

Nannotations
annotations
free_annotations
annotations is the record number of an array of GCardinals. Each element of the array is the record number of a GAnnotations structures. *Nannotations* is the number of elements allocated in the *annotations* array. *free_annotations* is the record number of the first free annotation, which forms the head of a linked list of free annotations.

Ntemplates
templates *templates* is the record number of an array of GCardinals. Each element of the array is the record number of a GTemplates structures. *Ntemplates* is the number of elements allocated in the *templates* array.

Nclones
clones *clones* is the record number of an array of GCardinals. Each element of the array is the record number of a GClones structures. *Nclones* is the number of elements allocated in the *clones* array.

Nvectors
vectors — *vectors* is the record number of an array of GCardinals. Each element of the array is the record number of a GVectors structures. *Nvectors* is the number of elements allocated in the *vectors* array.

contig_order
This is the record number of an array of GCardinals of size *NContigs*. Each element of the array is a contig number. The index of the array element indicates the position of this contig. Thus the contigs are displayed in the order that they appear in this array.

3.3.2 The GReadings Structure

```
/* GReadings.sense */
#define GAP_SENSE_ORIGINAL 0
#define GAP_SENSE_REVERSE  1
/* GReadings.strand */
#define GAP_STRAND_FORWARD 0
#define GAP_STRAND_REVERSE 1
/* GReadings.primer */
#define GAP_PRIMER_UNKNOWN 0
#define GAP_PRIMER_FORWARD 1
#define GAP_PRIMER_REVERSE 2
#define GAP_PRIMER_CUSTFOR 3
#define GAP_PRIMER_CUSTREV 4

/* GReadings.chemistry */
/* Bit 0 is 1 for terminator, 0 for primer */
#define GAP_CHEM_TERMINATOR (1<<0)
/* Bits 1 to 4 inclusive are the type (any one of, not bit pattern) */
#define GAP_CHEM_TYPE_MASK (15<<1)
#define GAP_CHEM_TYPE_UNKNOWN (0<<1)
#define GAP_CHEM_TYPE_ABI_RHOD (1<<1)
#define GAP_CHEM_TYPE_ABI_DRHOD (2<<1)
#define GAP_CHEM_TYPE_BIGDYE (3<<1)
#define GAP_CHEM_TYPE_ET (4<<1)
#define GAP_CHEM_TYPE_LICOR (5<<1)

typedef struct {
    GCardinal name;
    GCardinal trace_name;
    GCardinal trace_type;
    GCardinal left; /* left neighbour */
    GCardinal right; /* right neighbour */
    GCardinal position; /* position in contig */
    GCardinal length; /* total length of reading */
    GCardinal sense; /* 0 = original, 1 = reverse */
    GCardinal sequence;
    GCardinal confidence;
    GCardinal orig_positions;
    GCardinal chemistry; /* see comments above (GAP_CHEM_*) */
    GCardinal annotations; /* start of annotation list */
    GCardinal sequence_length; /* clipped length */
```

```
        GCardinal start; /* last base of left cutoff */
        GCardinal end; /* first base of right cutoff */
        GCardinal template; /* aka subclone */
        GCardinal strand; /* 0 = forward, 1 = reverse */
        GCardinal primer; /* 0 = unknown, 1 = forwards, */
    /* 2 = reverse, 3 = custom forward */
                                    /* 4 = custom reverse */
        GCardinal notes; /* Unpositional annotations */
    } GReadings;
```

The reading structure contains information related to individual sequence fragments. It should be read and written using the `gel_read` and `gel_write` functions. Whilst it is perfectly possible to use `GT_Read` to access this data, using `gel_read` will read from an in-memory cache and so is much faster. Using `GT_Write` to write a *GReadings* structure must never be used as it will invalidate the cache.

name The record number of the text string containing the reading identifier. Care must be taken to use the correct functions to access the reading name. Use `io_read_reading_name` and `io_write_reading_name` instead of `io_read_text` or `io_write_text`. See Section 2.2.3.9 [io_read_reading_name and io_write_reading_name], page 22.

trace_name
The record number of the text string containing the trace filename.

trace_type The record number of the text string containing the type of the trace.

left The left hand neighbour of this sequence, or 0 if this is the first reading in the contig. Sequences are stored in a doubly linked list which is sorted on positional order. The right hand neighbour of the sequence referenced by this field should be the same as this sequence number. NOTE: this is the reading number, not the record number.

right The right hand neighbour of this sequence, or 0 if this is the last reading in the contig. The left hand neighbour of the sequence referenced by this field should be the same as this sequence number. NOTE: this is the reading number, not the record number.

position The absolute position of this reading within the contig (starting from 1).

length The total length of this reading, including cutoff data.

sense The orientation of this reading. 0=original, 1=reversed. The `GAP_SENSE_*` macros should be used in preference to integer values.

sequence The record number of the text string containing the complete sequence.

confidence The record number of the 1 byte integer array containing the confidence values. This has one value per sequence base and so is the same length as the sequence array.

orig_positions
The record number of the 2 byte integer array containing the original positions of each base. This has one 2 byte value per sequence base.

chemistry The chemistry type of this reading. 0=normal. `chemistry & GAP_CHEM_DOUBLE` contains the terminator reaction information. Non zero implies a terminator reaction, which can then optionally be used as double stranded sequence.

annotations
: The number of the first annotation for this reading. Annotations are stored in a linked list structure. This value is 0 if no annotations are available. NOTE: This is not the same as the record number of the first annotation.

sequence_length
: The used length of sequence. This should always be the same as the *end-start-1*.

start The position of the last base in the left hand cutoff data (starting from 1).

end The position of the first base in the right hand cutoff data (starting from 1).

template The template number. Readings sharing a template (ie insert) have the same template number.

strand The strand this sequence was derived from. 0=forward, 1=reverse. The `GAP_STRAND_*` macros should be used in preference to integer values.

primer The primer type for this sequence. 0=unknown, 1=forward, 2=reverse, 3=custom forward, 4=custom reverse. The `GAP_PRIMER_*` macros should be used in preference to integer values.

3.3.3 The GContigs Structure

```
typedef struct {
    GCardinal left; /* left reading number */
    GCardinal right; /* right reading number */
    GCardinal length; /* contig sequence length */
    GCardinal annotations; /* start of annotation list */
    GCardinal notes; /* Unpositional annotations */
} GContigs;
```

left The number of the leftmost reading in this contig. This is a reading number, not a record number.

right The number of the rightmost reading in this contig. This is a reading number, not a record number. Note that the rightmost reading is defined as the reading the left end furthest to the right and not the reading with the right end furthest to the right.

length The total length of this contig.

annotations
: The annotation number of the first annotation on the consensus for this contig or 0 if none are available.

3.3.4 The GAnnotations Structure

```
typedef struct {
    GCardinal type;
    GCardinal position;
    GCardinal length;
    GCardinal strand;
    GCardinal annotation;
    GCardinal next;
} GAnnotations;
```

The annotations (aka tags) are comments attached to segments of readings or contig consensus sequences. The location is stored as position and length in the original orientation, so complementing a reading does not require edits to the annotations. Consensus sequences are always considered uncomplemented and so complementing a contig does require complementing of annotations that are stored on the consensus.

The annotations can be linked together to form linked lists, sorted on ascending position. The *GReadings* and *GContigs* structures contain an annotations field which holds the annotation number of the left most (original orientation) annotation.

Unused annotations are kept in an unsorted linked list referenced by the *free_annotatons* field of the *GDatabase* structure.

type The type of the annotation; a 4 byte integer which the user sees as a 4 character string.

position The position of the left end of the annotation.

length The length of the annotation.

strand The annotation strand. 0 for positive, 1 for negative, and 2 for both.

annotation
The record number of the text string containing a comment for the annotation. Zero means no comment.

next The annotation number of the next annotation in the linked list, or zero if this is the last in this linked list.

3.3.5 The GVectors Structure

```
/* GVectors.level */
#define GAP_LEVEL_UNKNOWN  0
#define GAP_LEVEL_CLONE    1
#define GAP_LEVEL_SUBCLONE 2

typedef struct {
    GCardinal name; /* vector name */
    GCardinal level; /* 1=clone, 2=subclone, etc */
} GVectors;
```

The vector structure contains simply information on any vectors used in cloning and sub-cloning. The *GTemplates* and *GClones* structures reference this structure.

name The record number of the text string containing the name for this vector.

level A numeric value for the level of the vector. Use the `GAP_LEVEL_*` macros for this field.

3.3.6 The GTemplates Structure

```
typedef struct {
    GCardinal name;
    GCardinal strands;
    GCardinal vector;
    GCardinal clone;
    GCardinal insert_length_min;
    GCardinal insert_length_max;
} GTemplates;
```

The template structure holds information about the physcial insert of a clone. A reading is within any single template, but several readings may share the same template.

name The record number of the text string containing the template name

strands The number of strands available. Either 1 or 2.

vector The vector number of the vector ("sequencing vector") used.

clone The clone number of the clone that this template came from.

insert_len_min
The minimum expected size of insert.

insert_len_max
The maximum expected size of insert.

3.3.7 The GClones Structure

```
typedef struct {
    GCardinal name;
    GCardinal vector;
} GClones;
```

The clone structure holds simple information to identify which original piece of materal our templates were derived from. Often we have a single clone per database and the database name is the same as the clone name.

name The record number of the text string containing the clone name.

vector The vector number of the vector used. The vector referenced here could be M13 for a very small project, or a cosmid, YAC or BAC for a larger "subcloned

3.4 The GapIO Structure

The main object passed around between the I/O functions is the *GapIO* structure. This is returned from the `open_db` function and is then passed around in much the same manner as a unix file descriptor or *FILE* pointer is. The structure, held in '`gap4/IO.h`', is as follows.

```
typedef struct {
    GapServer *server; /* our server */
    GapClient *client; /* ourselves */

    int Nviews; /* number of locked views */
    Array views; /* all locked views */

    GDatabase db; /* main database record */
    Bitmap freerecs; /* bitmap of unused */
    Array contigs; /* list of contig */
    Array readings; /* list of reading records */
    Array annotations; /* list of annotation records */
    Array templates; /* list of template records */
    Array clones; /* list of clone records */
    Array vectors; /* list of vector records */

    int4 *relpos; /* relpg[] */
    int4 *length; /* length[] */
    int4 *lnbr; /* lnbr[] */
    int4 *rnbr; /* rnbr[] */

    char db_name[DB_FILELEN]; /* database "file.version" */
```

```
        Array contig_order; /* order of contigs */
        Array contig_reg; /* Registration arrays for each contig */

    #ifdef GAP_CACHE
        Array reading; /* Array of GReading _structures_ */
        Array read_names; /* Array of reading names */
    #endif
        int freerecs_changed; /* Whether to flush freerecs bitmap */
        Bitmap updaterecs; /* bitmap of updated records */
        Bitmap tounlock; /* bitmap of records to unlock at next flush */
    } GapIO;
```

Many of the items held within this structure are used internally by the I/O functions. However it's worth describing all very briefly.

server
client The *server* and *client* pointers are used in the low level g library communication. They need not be used by any external code.

Nviews
views Each record in the database needs to be locked before it can be accessed. A view is returned for each independent lock of a record. These are used internally by the low level reading and writing function.

db This is a direct copy of the *GDatabase* structure for this database. This needs to be kept up to date with the on disk copy whenever changes are made (eg by adding a new reading).

freerecs This is a copy of the free records bitmap referenced by the *io->db.freerecs* field. It is kept up to date internally.

contigs
readings
annotations
templates
clones
vectors These are lookup arrays to convert structure numbers to record numbers. For instance, all readings are numbered from 1 upwards. Similarly for contigs. However reading number 1 and contig number 1 will have their own unique record numbers in the g database.

The extensible array package is used for storing this information. To translate from reading number N to the record number use "`arr(GCardinal, io->readings, N-1)`".

relpos
length
lnbr
rnbr These are arrays of 4-byte integers of size *io->db.actual_db_size*. They hold information about both readings and contigs.

For readings, the array contents hold copies of the *position*, *sequence_length*, *left* and *right* fields of the *GReadings* structures. Reading number R has this data stored in array elements R (counting from element 0, which is left blank).

For contigs, the array contents hold copies of the *length*, *left* and *right* fields of the *GContigs* structure. For historical reasons the contig length is held in the *relpos* array with the *length* array left blank. Contig number *C* has this data stored in array elements *io->db.actual_db_size-C*.

For ease of use and future compatibility several macros have been defined for accessing this data. See Section 3.5 [IO.h Macros], page 68. These should be used instead of direct access. Thus to find the length of reading *R* we use `io_length(io,R)` and to find the length of contig *C* we use `io_clength(io,C)`.

NOTE: These arrays are not updated automatically. If you modify data using one of the write functions you also need to update the arrays in sync. This is one of the problems that the check database command looks for so mistakes should be obvious.

db_name The name of the database in a *file.version* syntax. This array is allocated to be `DB_FILELEN` bytes long. The `io_name` macro should be used for accessing this field.

contig_order
An array loaded from *io->db.contig_order*. This holds the left to right ordering of contigs. It is automatically undated by the create and delete contig function.

contig_reg The contig registration scheme information. There's an entire chapter on this topic. See Chapter 6 [Gap4 Contig Registration Scheme], page 99.

reading
read_names
These are cached copies of the *GReadings* structures and the reading names referenced by the *GReadings.name* fields. They are updated automatically when using the correct functions (`gel_read` and `gel_write`). Use of lower level functions is disallowed for accessing this data.

freerecs_changed
updaterecs
tounlock These three are used internally for maintaining the update and data flushing scheme. *freerecs_changed* is a flag to state whether or not the *freerecs* bitmap needs writing to disk. *updaterecs* and *tounlock* are bitmaps with a bit per record to signify whether the record needs rewriting or unlocking. Their use is not required outside of the low level functions.

3.5 IO.h Macros

There are many C macros defined to interact with the *GapIO* structure. These both simplify and improve readeability of the code and also provide a level of future proofing. Where the macros are available it is always advisable to use these instead of accessing the *GapIO* structure directly.

Note that not all of these macros are actually held within the '`IO.h`' file, rather some are in files included by '`IO.h`'. However whenever wishing to use one of these macros you should still use `"#include <IO.h>"`.

`io_dbsize(`*io*`)`
io`->db.actual_db_size`

The maximum number of readings plus contigs allowed.

`max_gel_len(`*io*`)`
(io)`->max_gel_len`

The maximum reading length.

`NumContigs(`*io*`)`
(io)`->db.num_contigs`

The number of used contigs.

`NumReadings(`*io*`)`
: *(io)*`->db.num_readings`

  The number of used readings.

`Ncontigs(`*io*`)`
: *(io)*`->db.Ncontigs`

  The number of allocated contigs.

`Nreadings(`*io*`)`
: *(io)*`->db.Nreadings`

  The number of allocated readings.

`Nannotations(`*io*`)`
: *(io)*`->db.Nannotations`

  The number of allocated annotations.

`Ntemplates(`*io*`)`
: *(io)*`->db.Ntemplates`

  The number of annotated templates.

`Nclones(`*io*`)`
: *(io)*`->db.Nclones`

  The number of allocated clones.

`Nvectors(`*io*`)`
: *(io)*`->db.Nvectors`

  The number of allocated vectors.

`io_relpos(`*io,g*`)`
: *(io)*`->relpos[(`*g*`)]`

  The position of a reading *g*.

`io_length(`*io,g*`)`
: *(io)*`->length[(`*g*`)]`

  The length of a reading *g*. If the reading is complemented this value is negative, but still represents the length.

`io_lnbr(`*io,g*`)`
: *(io)*`->lnbr[(`*g*`)]`

  The reading number of the left neighbour of reading *g*.

`io_rnbr(`*io,g*`)`
: *(io)*`->rnbr[(`*g*`)]`

  The reading number of the right neighbour of reading *g*.

`io_clength(`*io,c*`)`
: *(io)*`->relpos[io_dbsize(`*io*`)-(`*c*`)]`

  The length of contig *c*.

`io_clnbr(`*io,c*`)`
: *(io)*`->lnbr[io_dbsize(`*io*`)-(`*c*`)]`

  The leftmost reading number of contig *c*.

`io_crnbr(`*io,c*`)`
: *(io)*`->rnbr[io_dbsize(`*io*`)-(`*c*`)]`

  The rightmost reading number of contig *c*.

`io_name(`*io*`)`
: *(io)*`->db_name`

  The database name.

`io_rdonly(`*io*`)`
: This returns 1 when the database has been opened as read-only; 0 otherwise.

`io_rname(`*io,g*`)`
: This returns the reading name for reading number *g*. This is fetched from the in memory cache.

`io_wname(`*io,g,n*`)`
: Sets the in-memory copy of the reading name for reading number *g* to be the string *n*. This does not write to disk.

`PRIMER_TYPE(`*r*`)`
: This returns the type of the primer used for sequencing reading number *r*. This information is calculated from the *primer* and *strand* fields of the *GReadings* structure. It returns one of `GAP_PRIMER_UNKNOWN`, `GAP_PRIMER_FORWARD`, `GAP_PRIMER_REVERSE`, `GAP_PRIMER_CUSTFOR` and `GAP_PRIMER_CUSTREV`.

`PRIMER_TYPE_GUESS(`*r*`)`
: As `PRIMER_TYPE` except always choose a sensible guess in place of `GAP_PRIMER_UNKNOWN`.

`STRAND(`*r*`)`
: Returns the strand (one of `GAP_STRAND_FORWARD` or `GAP_STRAND_REVERSE`) from the primer information for reading number *r*. The reason for these primer and strand macros is that the meaning of the *primer* and *strand* fields of *GReadings* has changed slightly from early code in that we now make a distinction between custom forward primers and custom reverse primers. The *strand* field may become completely redundant in future as it can now be derived entirely from the primer.

`contig_read(`*io, cn, c*`)`
`gel_read(`*io, gn, g*`)`
`tag_read(`*io, tn, t*`)`
`vector_read(`*io, vn, v*`)`
`clone_read(`*io, cn, c*`)`
: Reads one of the basic database structures. For contigs, `contig_read` reads contig number *cn* and stores in the *GContigs* structure named *c*. Eg to read the a contig:

```
GContigs c;
contig_read(io, contig_num, c);
```

  This is functionally equivalent to:

```
GContigs c;
GT_Read(io, arr(GCardinal, io->contigs, contig_num-1),
&c, sizeof(c), GT_Contigs);
```

  The exception to this is `gel_read` which reads from a cached copy held in memory.

`contig_write(`*io, cn, c*`)`
`gel_write(`*io, gn, g*`)`
`tag_write(`*io, tn, t*`)`
`vector_write(`*io, vn, v*`)`
`clone_write(`*io, cn, c*`)`
: Writes one of the basic types in a similar fashion to the read functions. To write to annotation number *anno* we should use:

```
GAnnotations a;
/* ... some code to manipulate 'a' ... */
tag_write(io, anno, a);
```

This is functionally equivalent to:

```
GT_Write(io, arr(GCardinal, io->annotations, anno-1),
 &a, sizeof(a), GT_Annotations);
```

Note that the `gel_write` function **must** be used instead of `GT_Write` as `gel_write` will also update the reading memory cache.

3.6 Basic Gap4 I/O

These functions consist of both basic functions for reading, writing and creation of database items and simple I/O functions that build upon such operations. They are mainly contained within the '`Gap4/IO.c`' file.

The return codes do vary greatly from function to function. Most return 0 for success and -1 for failure. However some will return other codes. In general it is best to check equality to the success code rather than equality to a specific failure code.

and read as an array of *GCardinal*s. *elements* indicates the number of array elements and not the size of the array in bytes.

`BitmapRead` reads records of type `GT_Bitmap`. The bitmap is allocated by this function. *elements* indicates the number of bits and not the size of the bitmap in bytes.

3.6.1 GT_Write, GT_Write_cached, TextWrite, DataWrite, ArrayWrite, BitmapWrite

```
#include <IO.h>

int GT_Write(
        GapIO  *io,
        int     rec,
        void   *buf,
        int     len,
        GCardinal type);

int GT_Write_cached(
        GapIO  *io,
        int     read,
        GReadings *r);

int TextWrite(
        GapIO  *io,
        int     rec,
        char   *buf,
        int     len);

int DataWrite(
        GapIO  *io,
        int     rec,
        void   *buf,
        int     len,
        int     size);
int ArrayWrite(
        GapIO  *io,
        int     rec,
        int     elements,
        Array   a);

int BitmapWrite(
        GapIO  *io,
        int     rec,
        Bitmap  b);
```

These functions write record number *rec* with the appropriate data type. They return zero for success and an error code for failure.

`GT_Write` writes arbitrary records of type *type*. This is usually a structure. Do not use this function for writing *GReadings* structures. For best compatibility, use the `contig_write`, `gel_write`, `tag_write`, `vector_write` and `clone_write` function.

`GT_Write_cached` is an interface to `GT_Write` which also updates the in-memory reading cache. For best compatibility, use the `gel_write` function.

`TextWrite` writes a record of type `GT_Text`. It is used to write text only strings.

`DataWrite` writes a record of type `GT_Data`. It is used to write binary data such as sequence confidence values.

`ArrayWrite` writes a record of type `GT_Array`. The array must be an array of *GCardinal* values. *elements* indicates the number of array elements and not the size of the array in bytes.

`BitmapWrite` writes a record of type `GT_Bitmap`. *elements* indicates the number of bits and not the size of the bitmap in bytes.

3.6.2 io_handle and handle_io

```
#include <IO.h>

GapIO *io_handle(
        f_int *handle);

f_int *handle_io(
        GapIO *io);
```

These two routines convert between *GapIO* pointers and integer handles. Both the Fortran and Tcl code uses integer handles due to no support for structures.

`io_handle` takes a pointer to an integer handle and returns the associated *GapIO* pointer. It returns NULL for failure.

`handle_io` takes a *GapIO* pointer and returns a pointer to a integer handle. It returns NULL for failure.

3.6.3 io_read_seq

```
#include <IO.h>

int io_read_seq(
        GapIO  *io,
        int     N,
        int2   *length,
        int2   *start,
        int2   *end,
        char   *seq,
        int1   *conf,
        int2   *opos);
```

This function loads from memory and disk information on gel readings and stores this in the paramaters passed over.

The reading number to read should be passed as *N*. The integers pointed to by *length*, *start* and *end* pointers are then written to with the total length (*GReadings.length*), the last base number (counting from 1) of the left hand cutoff data, and the first base number of te right hand cutoff data.

The sequence, confidence and original position data is then loaded and stored in the address pointed to by *seq*, *conf* and *opos* respectively. This is expected to be allocated to the correct size by the caller of this function. Either or both of *conf* and *opos* can be NULL, in which case the data is not loaded or stored. *seq* must always be non NULL.

This function returns 0 for success and non zero for failure.

3.6.4 io_write_seq

```
#include <IO.h>


int io_write_seq(
        GapIO  *io,
        int     N,
        int2   *length,
        int2   *start,
        int2   *end,
        char   *seq,
        int1   *conf,
        int2   *opos);
```

This function updates disk and memory details of reading number *N*. If this reading does not yet exist, all non existant readings up to and including *N* will be initialised first using the `io_init_readings` function.

[FIXME: The current implement **does not** update the fortran lngth (io_length()) array. This needs to be done by the caller.]

The *length* argument is the total length of the sequence, and hence also the expected size of the *seq*, *conf* and *opos* arrays. *start* and *end* contain the last base number of the left cutoff data and the first base number of the right cutoff data.

Unlike *io_read_seq*, all arguments to this function are mandatory. If the records on disk do not already exist then they are allocated first using the `allocate` function.

This function returns 0 for success and non zero for failure.

3.6.5 get_read_info, get_vector_info, get_clone_info and get_subclone_info

```
#include <IO.h>

int get_read_info(
        GapIO  *io,
        int     N,
        char   *clone,
        int     l_clone,
        char   *cvector,
        int     l_cvector,
        char   *subclone,
        int     l_subclone,
        char   *scvector,
        int     l_scvector,
        int    *length,
        int    *insert_min,
        int    *insert_max,
        int    *direction,
        int    *strands,
        int    *primer,
        int    *clone_id,
        int    *subclone_id,
        int    *cvector_id,
        int    *scvector_id);

int get_vector_info(
        GapIO  *io,
        int     vector_id,
        char   *vector,
        int l_vector);

int get_clone_info(
        GapIO  *io,
        int     clone_id,
        char   *clone,
        int     l_clone,
        char   *cvector,
        int     l_cvector,
        int    *cvector_id);

int get_subclone_info(
        GapIO  *io,
        int     subclone_id,
        char   *clone,
        int     l_clone,
        char   *cvector,
        int     l_cvector,
        char   *subclone,
        int     l_subclone,
        char   *scvector,
        int     l_scvector,
        int    *insert_min,
        int    *insert_max,
        int    *strands,
        int    *clone_id,
        int    *cvector_id,
            int    *scvector_id);
```

These functions return clone, template and vector information.

`get_vector_info` returns the name of a vector. This is stored in the buffer at *vector*.

`get_clone_info` function returns the name of the clone and the vector number (stored at *clone* and *cvector_id* and results of `get_vector_info` for this vector.

`get_subclone_info` returns the template information (insert size, number of strands, vector and clone numbers stored at *insert_min*, *insert_max*, *strands*, *scvector_id* and *clone_id*) along with the results from `get_vector_info` and `get_clone_info` on the appropriate vector and clone numbers.

`get_read_info` returns the reading information including direction, primer, template (subclone) number (stored at *direction*, *strands*, *primer*, and *clone_id*), and the results of the `get_subclone_info` on this template number.

For all four functions, the arguments used to store text fields, such as the clone name (*clone*), all have corresponding buffer lengths sent as the same argument name preceeded by *l_* (eg *l_clone*). These buffers need to be allocated by the caller of the function.

Any buffer or integer pointer arguments may be passed as `NULL` to avoid filling in this field. For buffers the same is also true when specifying the buffer length as zero.

The *clone*, *vector* and *subclone* buffers are used to store the names of the clone, vector or template. If appropriate, the clone or template number will also be stored at the *clone_id* and *subclone_id* addresses.

For functions returning information more than one vector, these are split into two levels. The sequencing vector is the vector used to sequence this template. It has arguments named *scvector* (name), *l_scvector* (name length) and *scvector_id* (vector number). The clone vector is the vector used in the sequecing of the fragment which is later broken down and resequenced as templates. This may not be appropriate in many projects. It has arguments named *cvector* (name), *l_cvector* (name length) and *cvector_id* (vector number).

All functions return 0 for success and an error code for failure.

3.6.6 io_init_reading, io_init_contig and io_init_annotations

```
#include <IO.h>

int io_init_reading(
        GapIO  *io,
        int     N);

int io_init_contig(
        GapIO  *io,
        int     N);

int io_init_annotations(
        GapIO  *io,
        int     N);
```

These functions create new reading, contig and annotations structures. Each takes two arguments; the first being the *GapIO* pointer, and the second being the new reading, contig or annotation number to create. This is not the number of new structures, but rather the highest allowed number for this structure. For instance, if we have 10 readings, `"io_init_reading(io, 12)"` will create two more, numbered 11 and 12.

For readings, the records are recovered (by increasing the *GDatabase NumReadings* field to *NReadings*) if available. The new *GReadings* structure are not guaranteed to be clear.

For contigs, the records are recovered if available. The contig_order array is also updated with the new contigs being added at the rightmost position. The new contigs are added to the registration scheme with blank registration lists. The new *GContigs* structures are not guaranteed to be clear.

For annotations, new records are always allocated from disk. It is up to the caller to first check that there are no free annotations in the *free_annotations* list. The new *GAnnotations* structures are not guaranteed to be clear.

All functions returns return 0 for success, and -1 for failure.

3.6.7 io_read_annotation and io_write_annotation

```
#include <IO.h>

int io_read_annotation(
        GapIO  *io,
        int     N,
        int    *anno);

int io_write_annotation(
        GapIO  *io,
        int     N,
        int    *anno);
```

These functions read and write the first annotation number in the linked lists referenced by the reading and contig structures. For both functions, *N* is a reading number if it is above zero or a contig number when below zero (in which case it is negated).

`io_read_annotation` reads the *annotations* field of reading *N* or contig -*N* and stores this in *anno*. It sets *anno* to 0 returns 1 for failure. Otherwise it returns 0.

`io_write_annotation` sets the *annotations* field of reading *N* or contig -*N* to be **anno*. Despite the fact that it is a pointer, the contents of *anno* is not modified. It returns 1 for failure and 0 for success (but currently always returns 0).

3.6.8 allocate

```
#include <IO.h>

int allocate(
        GapIO    *io,
        GCardinal type);
```

These allocate and deallocate records in the g database.

Th `allocate` function allocates a new record from the g database. It finds a free record, or creates a new record, and returns this record number. The record will be automatically locked for exclusive read/write access. The type of the record is sent in *type*. This must be one of following:

`GT_Text`

`GT_Data`

```
GT_Array
GT_Bitmap
GT_Database
GT_Contigs
GT_Readings
GT_Vectors
GT_Annotations
GT_Templates
GT_Clones
```

The function does not initialise or even write the new record to disk. The record number is valid, but a `GT_Read` call will produce an error. It is up to the caller to initialise the structure and perform the first `GT_Write` (or equivalent) call.

It returns the record number for success, and terminates the program for failure.

3.6.9 deallocate

```
#include <IO.h>

int deallocate(
        GapIO    *io,
        int       rec);
```

The `deallocate` function removes record *rec* from the g database. This uses the `g_remove` function, but unlocking is only performed at the next database flush.

It returns 0 for success and 1 for failure.

3.6.10 io_deallocate_reading

```
#include <IO.h>

int io_deallocate_reading(
        GapIO  *io,
        int     N);
```

The `io_deallocate_reading` function deallocates the records linked to by reading number *N*. These are the *name*, *trace_name*, *trace_type*, *sequence*, *confidence* and *orig_positions* fields of the *GReadings* structure.

The reading itself is not deallocated. The operation of Gap4 requires that reading numbers are sequential with all numbers used. It is up to the caller of this routine to make sure that this is still true.

It returns 0 for success and >=1 for failure.

3.6.11 io_read_rd and io_write_rd

```
#include <IO.h>

int io_read_rd(
        GapIO  *io,
        int     N,
        char   *file,
        int     filelen,
        char   *type,
        int     typelen);

int io_write_rd(
        GapIO  *io,
        int     N,
        char   *file,
        int     filelen,
        char   *type,
        int     typelen);
```

These routines read and write the reading 'raw data' paramaters. These are the file name and file type of the sequence trace file.

For both functions, *N* is the reading number; *file* is a buffer, allocated by the caller, of length *filelen*; and *type* is a buffer, allocated by the caller, of length *typelen*.

`io_read_rd` copies the trace filename to *file* and it's type to *type*. If either of these unknown the corresponding buffer is filled with spaces instead. It returns 0 if both name and type are known and 1 is either or both are unknown.

`io_write_rd` write new file name and file type information. If *N* is an unknown reading number, it is first allocated using `io_init_readings`. It returns 0 for success.

3.6.12 open_db

```
#include <IO.h>

GapIO *open_db(
        char   *project,
        char   *version,
        int    *status,
        int     create,
        int     read_only);
```

`open_db` opens existing databases or creates new databases. The database to be opened or created has unix filenames of "*project*.*version*" and "*project*.*version*.aux".

The *create* variable should be 0 or 1. A value of 1 indicates that this database is to be created. This will not be done if there is a file named "*project*.*version*.BUSY", in which case the *status* variable is set to contain `IO_READ_ONLY`.

The *read_only* variable should be 0 or 1. A value of 1 indicates that the database should be opened in read only mode, otherwise read/write access is desired. If the database is busy then the database may still be opened in read only mode instead. In this case the *status* variable is set to contain `IO_READ_ONLY`.

The *GapIO* structure is then initialised and returned. A successful return will leave *status* containing 0. For failure, the function returns NULL.

3.6.13 close_db

```
#include <IO.h>

int close_db(
        GapIO *io);
```

This function closes a database. *io* is a *GapIO* pointer returned from a previous call to `open_db`. If necessary, the busy file is removed, and all allocated memory is freed.

The function returns 0 for success and -1 for failure.

3.6.14 del_db

```
#include <IO.h>

int del_db(
        char   *project,
        char   *version);
```

This removes the databases files for a particular *version* of a *project*. The database should not be open at the time of calling this function. On unix, the files removed are named "*project.version*" and "*project.version*.aux".

3.6.15 flush2t

```
#include <IO.h>

void flush2t(
        GapIO *io);
```

This functions checks out all written data by updating the database time stamp. If Gap4 crashes, upon restarting any data written since the last time stamp is ignored. The purpose of this is to ensure that the data in the database is internally consistent. Hence you should only call this function when the database writes are consistent.

An example of this is in deleting a reading *N* which has left and right neighbours of *L* and *R*. The operation of writes may be:

- set right neighbour of *L* to be *R*
- set left neighbour of *R* to be *L*
- deallocate *N*.

The database is consistent before these operations, and after these operations, but not at any stage in between.

3.6.16 get_gel_num and get_contig_num

```
#include <IO.h>

int get_gel_num(
        GapIO  *io,
        char   *gel_name,
        int     is_name);

int get_contig_num(
        GapIO  *io,
        char   *gel_name,
        int     is_name);
```

These functions convert reading and contig names into reading and contig numbers. (A contig name is defined to be the name of any reading held within that contig.)

The *is_name* argument is mainly used for backwards compatibility. It should be passed as either `GGN_ID` or `GGN_NAME`. When equal to `GGN_ID`, *gel_name* is treated as a *reading identifier*, otherwise it is treated as a *reading name*. An identifier is defined to be either a reading name; a hash sign followed by a reading number; or an equals sign followed by a contig number.

Both functions return -1 for failure or the appropriate reading or contig number for success.

3.6.17 lget_gel_num and lget_contig_num

```
#include <IO.h>

int lget_gel_num(
        GapIO  *io,
        int     listArgc,
        char  **listArgv,
        int    *rargc,
        int   **rargv);

int lget_contig_num(
        GapIO  *io,
        int     listArgc,
        char  **listArgv,
        int    *rargc,
        contig_list_t **rargv);
```

These functions perform the same task as `get_gel_num` and `get_contig_num` except on lists of identifier instead of single identifiers.

The list of identifiers is passed in *listArgv* as an array of *listArgc* strings. They return arrays of reading or contig numbers by setting **rargv* to point to an array of **rargc* elements. The memory is allocated by these functions and should be deallocated by the caller using `free`.

For `lget_gel_num` the return arrays are arrays of integer values. `lget_contig_num` returns arrays of *contig_list_t* structures. This structure is defined as follows.

```
typedef struct contig_list {
    int contig;
    int start;
    int end;
} contig_list_t;
```

If any string within the *listArgv* argument to `lget_contig_num` is a list, the second and third elements of this list are used to define the *start* and *end* offsets within the contig (which is defined by the name held in the first element of the list). Otherwise, the *start* and *end* fields are set to 1 and the length of the contig respectively.

For instance, it is legal for pass over `"rname"`, `"rname 100"` and `"rname 100 200"` as contig identifiers.

Both functions return 0 for success and -1 for failure. Note that the returned *rargc* value may not be the same as *listArgc* in the case where one or more identifiers could not be translated.

3.6.18 to_contigs_only

```
#include <IO.h>

int *to_contigs_only(
        int     num_contigs,
        contig_list_t *cl);
```

This functions converts an array of *contig_list_t* structures to an array of integers containing only the contig number information. The *cl* and *num_contigs* elements correspond to the returned *rargv* and *rargc* arguments from the `lget_contig_num` function.

It returns a malloc array of integers for success or `NULL` for failure.

3.6.19 chain_left

```
#include <IO.h>

int chain_left(
        GapIO  *io,
        int     gel);
```

This function finds the left most reading number of the contig containing the reading numbered *gel*. This is done by chaining along the left neighbours of each reading in turn until the contig end is reached. The function detects possible loops and returns -1 in this case. Otherwise the left most reading number is returned.

3.6.20 rnumtocnum

```
#include <IO.h>

int rnumtocnum(
        GapIO  *io,
        int     gel);
```

This function returns the contig number for the contig containing the reading numbered *gel*. It returns -1 if the contig number cannot be found.

3.7 Other I/O functions

This section includes all the other I/O functions which don't fit well into the other sections. Specifically, these functions cannot be used when compiling external programs that utilise the gap4 I/O functions. The reason for this is that they require many other portions of the gap4 objects which in turn require more.

Whilst it is possible to still link in this manner, it is unwieldy and far from ideal. If you have need to use any of these functions in code that is to run separate from Gap4 then please mail us. We will then investigate tidying up the code further to aid such compilations.

3.7.1 io_get_extension

```
#include <IO.h>

int io_get_extension(
GapIO  *io,
int N,
char   *seq,
int max_seq,
int    *length,
int    *complement);
```

`io_get_extension` reads the usable 3' cutoff data for reading number *N*. The cutoff data is stored in *seq*. The length stored is the smaller of *max_seq* bytes or the length of the 3' cutoff data. The length of data stored in *seq* is written to the *length* pointer. The orientation of the reading is stored in the *complement* pointer.

The reading annotations are also read to determine which segments are considered usable. The existance of a tag with type `IGNS` or `IGNC`, anywhere on the reading, indicates that there is no suitable cutoff data for this reading. *length* is set to 1 and the function returns 1.

If a tag of type `SVEC` or `CVEC` exists within the 3' cutoff data the segment returned consists of that between the 3' cutoff point and the start of the vector tag.

The function returns 0 for success and 1 for failure.

NOTE: The current implementation looks any tags with types `IGN?` and `?VEC` rather than the specific types listed.

3.7.2 io_mod_extension

```
#include <IO.h>

int io_mod_extension(
GapIO  *io,
int N,
int shorten_by);
```

`io_mod_extension` modifies the position of the 3' cutoff data for reading number *N*. The 3' cutoff position is defined to be the base number, counting from 1, of the first base within the cutoff data.

shorten_by is subtracted from either the *end* or *start* field in the *GReadings* structure, depending on whether the reading is complemented. It is legal to specify a negative amount to increase the used portion of the reading.

[FIXME]

NOTE that this implementation does not set the *sequence_length* field or the `io_length(io,N)` data for this reading.

3.7.3 io_insert_base, io_modify_base and io_delete_base

```
#include <IO.h>

int io_insert_base(
GapIO  *io,
int gel,
int pos,
char base);

int io_modify_base(
GapIO  *io,
int gel,
int pos,
char base);

int io_delete_base(
GapIO  *io,
int gel,
int pos);
```

These functions modify readings by inserting, changing, or deleting individual bases. Where needed, they update any annotations on the reading to ensure that all annotations are still covering the same sequence fragments. The confidence values and original positions arrays are also updated. Inserted and edited bases are given confidence of 100 and original positions of 0.

`io_insert_base` uses the `io_insert_seq` function to inserts a single base with chacter *base* to at base position *pos*. Positions are measured counted such that inserting at base 1 inserts a base at the start of sequence.

`io_modify_base` uses the `io_replace_seq` function to replace a single base at position *pos* with *base*.

`io_delete_base` uses the `io_delete_seq` function to delete a single base at position *pos*.

FIXME:

NOTE that `io_insert_base` and `io_delete_base` modify the sequence, but DO NOT update the the *GReadings.sequence_length* or `io_length()` data.

3.7.4 io_delete_contig

```
#include <IO.h>

int io_delete_contig(
GapIO  *io,
int contig_num);
```

This function deletes a single contig number from the database. It **does not** remove any of the readings on the contig, but all annotations on the consensus sequence for this contig are deallocated.

The last contig in the database is renumbered to be *contig_num*. This updates the `io_clength()`, `io_clnbr()`, and `io_crnbr()` arrays in *io* and the contig order information.

A `REG_DELETE` notification is sent to the deleted contig **after** removal, followed by a `REG_NUMBER_CHANGE` notification to renumbered contig, followed by updating the contig registry tables.e

It returns 0 for success.

3.7.5 write_rname

```
#include <IO.h>

int write_rname(
GapIO  *io,
int rnum,
char   *name);
```

This writes a new reading name *name* for reading number *rnum*. This updates both the disk and memory copies of the reading structure and the reading name cache, using the `gel_write` and `io_wname` functions. If reading *rnum* does not exist, it is created first using the `io_init_reading` function.

It returns 0 for success and -1 for failure.

3.7.6 get_read_name, get_contig_name, get_vector_name, get_template_name, and get_clone_name

```
#include <IO.h>

char *get_read_name(
GapIO  *io,
int number);

char *get_contig_name(
GapIO  *io,
int number);

char *get_vector_name(
GapIO  *io,
int number);

char *get_template_name(
GapIO  *io,
int number);

char *get_clone_name(
GapIO  *io,
int number);
```

These functions convert reading, contig, vector, template and clone numbers into reading, contig, vector, and clone names respectively. Each function takes a *number* argument and returns a string containing the name. The string is held in a static buffer and is valid only until the next call of the same function. If the name is unknown, the string `"???"` is returned.

4 Sequence Editing Functions in C

4.1 io_complement_seq

```
#include <IO.h>

int io_complement_seq(
        int2   *length,
        int2   *start,
        int2   *end,
        char   *seq,
        int1   *conf,
        int2   *opos);
```

This function complements a sequence held in memory. No database I/O is performed. A sequence of length **length* is passed in the *seq* argument with associated confidence values (*conf*) and original positions (*opos*) arrays. The *start* and *end* arguments contain the left and right cutoff points within this sequence.

The function will reverse and comlement the sequence, negate the *start* and *end* values, and reverse the *conf* and *opos* arrays. If either of *conf* or *opos* are passed as `NULL`, neither will be reversed. *length* is not modified, despite the fact that it is passed by reference.

The function returns 0 for success.

4.2 io_insert_seq

```
#include <IO.h>

int io_insert_seq(
        int2    maxgel,
        int2   *length,
        int2   *start,
        int2   *end,
        char   *seq,
        int1   *conf,
        int2   *opos,
        int2    pos,
        char   *bases,
        int1   *newconf,
        int2   *newopos,
        int2    Nbases);
```

`io_insert_seq` inserts one or more bases into the sequence, confidence and original positions arrays specified. No database I/O is performed.

The existing sequence, confidence values, and original positions arrays are passed as *seq*, *conf*, and *opos* arguments. All are mandatory. The length of sequence and hence the number of used elements in these arrays is passed as *length*, with *start* and *end* containing the left and right cutoff positions.

The new sequence, confidence values, and original positions to insert are passed as *bases*, *newconf* and *newopos*. The number of bases to insert is *Nbases*. Either or both of the *newconf* and *newopos* arguments may be NULL. The inserted confidence values will then default to 100 for non pad (`"*"`) bases. For pads, the confidence value defaults to the average of the confidence values of the first two neighbouring bases that are not pads. The inserted original positions default to 0. These bases are to be inserted at the position specified by *pos*, counting as position 1 being to the left of the first base in the sequence.

As this operation increases the size of the *seq*, *conf*, and *opos* arrays, their allocated size must be passed in *maxgel*. If the insertion causes data to be shuffle beyond the *maxgel*, the right end of the sequence is clipped to ensure that no more than *maxgel* bases are present. The *start* and *end* values may be incremented, depending on where the insertion occurs.

This function returns 0 for success.

4.3 io_delete_seq

```
#include <IO.h>

int io_delete_seq(
        int2    maxgel,
        int2   *length,
        int2   *start,
        int2   *end,
        char   *seq,
        int1   *conf,
        int2   *opos,
        int2    pos,
        int2    Nbases);
```

`io_delete_seq` removes one or more bases from the sequence, confidence and original positions arrays specified. No database I/O is performed.

The existing sequence, confidence values, and original positions arrays are passed as *seq*, *conf*, and *opos* arguments. All are mandatory. The length of sequence and hence the number of used elements in these arrays is passed as *length*, with *start* and *end* containing the left and right cutoff positions. The allocated size of these arrays is *maxgel*, however it is not required by this function (FIXME).

The *pos* and *Nbases* arguments specify where and how many bases to delete, counting with the first base as base number 1. The *length* argument is described by *Nbases* and the *seq*, *conf* and *opos* arrays shuffled accordingly. The *start* and *end* values may be decrememnted, depending on where the deletion occurs.

The function returns 0 for success.

4.4 io_replace_seq

```
#include <IO.h>

int io_replace_seq(
        int2    maxgel,
        int2   *length,
        int2   *start,
        int2   *end,
        char   *seq,
        int1   *conf,
        int2   *opos,
        int2    pos,
        char   *bases,
        int1   *newconf,
        int2   *newopos,
        int2    Nbases,
        int     diff_only,
        int     conf_only);
```

`io_replace_seq` replaces on or more bases from the sequence, confidence and original positions arrays specified. No database I/O is performed.

The existing sequence, confidence values, and original positions arrays are passed as *seq*, *conf*, and *opos* arguments. All are mandatory. The length of sequence and hence the number of used elements in these arrays is passed as *length*, with *start* and *end* containing the left and right cutoff positions. The allocated size of these arrays is *maxgel*. FIXME: it is used - does it need to be?

The new sequence, confidence values, and original positions to replace are passed as *bases*, *newconf* and *newopos*. The number of bases to replace is *NBases*. Either or both of the *newconf* and *newopos* arguments may be NULL. The replaced confidence values will then default to 100 for non pad (`"*"`) bases. For pads, the confidence value defaults to the average of the confidence values of the first two neighbouring bases that are not pads. The replaced original positions default to 0. These bases are to be inserted at the position specified by *pos*, counting as position 1 being to the left of the first base in the sequence. The *length*, *start* and *end* values are left unchanged.

This function returns 0 for success.

4.5 pad_consensus

```
#include <IO.h>

int pad_consensus(
        GapIO  *io,
        int     contig,
        int     pos,
        int     npads);
```

This function inserts *npads* pads into the consensus for contig number *contig* at position *pos* by inserting into all of the readings creating the consensus at this point.

The function deals with inserting to the appropriate readings including adjustment of cutoff positions and annotations, moving of all the readings to the right of *pos*, and adjustment of the annotations on the consensus sequence.

It returns 0 for success.

4.6 calc_consensus

```
#include <qual.h>

int calc_consensus(
        int     contig,
        int     start,
        int     end,
        int     mode,
        char   *con,
        char   *con2,
        float  *qual,
        float  *qual2,
        float   cons_cutoff,
        int     qual_cutoff,
        int    (*info_func)(int          job,
                            void        *mydata,
                            info_arg_t  *theirdata),
        void   *info_data);

int database_info(
        int          job,
        void        *mydata,
        info_arg_t  *theirdata);
```

This function calculates the consensus sequence for a given segment of a contig. It can produce a single consensus sequence using all readings, or split it into two sequences; one for each strand. Additionally, it can produce either one (combinded strands) or two (individual strands) sets of values relating to the accuracy of the returned consensus.

The *contig*, *start* and *end* arguments hold the contig and range to calculate the consensus for. The ranges are inclusive and start counting with the first base as position 1.

con and *con2* are buffers to store the consensus. These are allocated by the caller to be at least of size *end-start+1*. If *con2* is `NULL` both strands are calculated as a single consensus to be stored in *con*. Otherwise the top strand is stored in *con* and the bottom strand is stored in *con2*.

mode should be one of `CON_SUM` or `CON_WDET`. `CON_SUM` is the "normal" mode, which indicates that the consensus sequence is simply the most likely base or a dash (depending on *cons_cutoff*. The `CON_WDET` mode is used to return special characters for bases that are good quality and identical on both strands. Where one strand has a dash, the consensus base for the other strand is used. Where both strands differ, and are not dashes, the consensus is returned as dash. Note that despite requiring the consensus for each starnd independently, this mode requires that *con2* is `NULL`. To summarise the action of the `CON_WDET` mode, the final consensus is derived as follows:

Top Strand	Bottom Strand	Resulting Base
A	A	d
C	C	e
G	G	f
T	T	i
-	-	-
-	*x*	*x*
x	-	*x*
x	*y*	-

[Where *x* and *y* are one of A, C, G or T, and *x* != *y*.]

qual_cutoff and *cons_cutoff* hold the quality and consensus cutoff paramaters used in the consensus algorithm for determining which bases are of sufficient quality to use and by how big a majority this base type must have before it is returned as the consensus base (otherwise "-" is used). For a complete description of how these parameters operate see the consensus algorithm description in the main Gap4 manual. (FIXME: should we duplicate this here?)

The *qual* and *qual2* buffers are allocated by the caller to be the same size as the *con* and *con2* buffers. They are filled with the a floating point representing the ratio of score for the consensus base type to the score for all base types (where the definition of score depends on the *qual_cutoff* parameter). This is the value compared against *cons_cutoff* to determine whether the consensus base is a dash. Either or both of *qual* and *qual2* can be passed as NULL if no accuracy information is required. Note that the accuracy information for *qual2* is only available when *con2* has also been passed as non NULL.

The algorithm uses *info_func* to obtain information about the readings from the database. *info_data* is passed as the second argument (*mydata*) to *info_func*. *info_func* is called each time some information is required about a reading or contig. It's purpose is to abstract out the algorithm from the data source. There are currently two such functions, the most commonly used of which is `database_info` function (the other being `contEd_info` to fetch data from the contig editor structures). The `database_info` function obtains the sequence details from the database. It requires a *GapIO* pointer to be passed as *info_data*.

The function returns 0 for success, -1 for failure.

4.7 calc_quality

```
#include <qual.h>

int calc_quality(
        int     contig,
        int     start,
        int     end,
        char   *qual,
        float   cons_cutoff,
        int     qual_cutoff,
        int   (*info_func)(int          job,
                           void        *mydata,
                           info_arg_t  *theirdata),
        void   *info_data)

int database_info(
        int          job,
        void        *mydata,
        info_arg_t  *theirdata);
```

This function calculates the quality codes for a given segment of a contig consensus sequence. The quality information is stored in the *qual* buffer, which should be allocated by the caller to be at least *end-start+1* bytes long. The contents of this buffer is one byte per base, consisting of a letter between 'a' and 'j'. There are #defines in '`qual.h`' assigning meanings to these codes, which should be used in preference to hard coding the codes themselves. The defines and meanings are as follows.

`a - R_GOOD_GOOD_EQ`
: Data is good on both strands and both strands agree on the same consensus base.

`b - R_GOOD_BAD`
: Data is good on the top strand, but poor on the bottom strand.

`c - R_BAD_GOOD`
: Data is good on the bottom strand, but poor on the top strand.

`d - R_GOOD_NONE`
: Data is good on the top strand, but no data is available on the bottom strand.

`e - R_NONE_GOOD`
: Data is good on the bottom strand, but no data is available on the top strand.

`f - R_BAD_BAD`
: Data is available on both strands, but both strands are poor data.

`g - R_BAD_NONE`
: Data is poor on the top strand, with no data on the bottom strand.

`h - R_NONE_BAD`
: Data is poor on the bottom strand, with no data on the top strand.

`i - R_GOOD_GOOD_NE`
: Data is good on both strands, but the consensus base differs between top and bottom strand.

`j - R_NONE_NONE`
: No data is available on either strand (this should never occur).

The *contig*, *start* and *end* arguments hold the contig and range to calculate the quality for. The ranges are inclusive and start counting with the first base as position 1.

qual_cutoff and *cons_cutoff* hold the quality and consensus cutoff parameters. These are used in an identical manner to the `calc_quality` function. See Section 4.7 [calc_quality], page 89.

The *info_func* and *info_data* arguments are also used in the same way as `calc_quality`. Generally *info_func* should be `database_info` and *info_data* should be a *GapIO* pointer. This will then read the sequence data from the Gap4 database.

The function returns 0 for success, -1 for failure.

5 Annotation Functions in C

NOTE: The terms *annotation* and *tag* are freely interchangable. Their varying use simply reflects the evolution of the code.

5.1 shift_contig_tags

```
#include <tagUtils.h>

void shift_contig_tags(
        GapIO  *io,
        int     contig,
        int     posn,
        int     dist);
```

This function moves tags within a contig with number *contig*. All tags starting at position *posn*, or to the right of *posn* are moved to the right by *dist* bases. *dist* should not be a negative value.

The function is used internally by (for example) algorithms to add pads to the consensus or for joing contigs.

5.2 merge_contig_tags

```
#include <tagUtils.h>

void merge_contig_tags(
        GapIO  *io,
        int     contig1,
        int     contig2,
        int     off);
```

This function is used to join a tag list from one contig to a tag list from another contig. All the tags in contig number *contig1* are added to contig number *contig2*. Each tag is moved by *off* bases when it is copied. The tag list is correctly maintained as sorted list. At the end of the function, the tags list for contig number *contig2* is set to 0.

The main purpose of this function is for use when joining contigs.

5.3 complement_contig_tags

```
#include <tagUtils.h>

void complement_contig_tags(
        GapIO  *io,
        int     contig);
```

This function complements the positions and orientations of each tag on the consensus sequence for contig number *contig*. The tags on the readings are not modified as these are always kept in their original orientation.

5.4 split_contig_tags

```
#include <tagUtils.h>

void split_contig_tags(
        GapIO  *io,
        int     cont1,
        int     cont2,
        int     posl,
        int     posr);
```

This function is called by the break contig algorithm and has little, if any, other use. When we're splitting a contig in half we need to move the annotations too. Annotations that overlap the two contigs are duplicated. Annotations that overlap the end of a contig have their lengths and positions corrected.

posl and *posr* hold the overlap region of contigs *cont1* and *cont2* before splitting. At the time of calling this routine, *cont2* has just been created (and has no tags). Both contigs have their lengths set correctly, but all of the tags are still in *cont1*. This function corrects these tag locations.

5.5 remove_contig_tags

```
#include <tagUtils.h>

void remove_contig_tags(
        GapIO  *io,
        int     contig,
        int     posl,
        int     posr);
```

This function removes annotations over the region defined as *posl* to *posr* from the consensus for contig number *contig*. Passing *posl* and *posr* as zero implies the entire consensus. This uses the `rmanno` function for the main portion of the work.

5.6 remove_contig_tags

```
#include <tagUtils.h>

void remove_gel_tags(
        GapIO  *io,
        int     gel,
        int     posl,
        int     posr);
```

This function removes annotations over the region defined as *posl* to *posr* from the reading numbered *gel*. Passing *posl* and *posr* as zero implies the entire reading. This uses the `rmanno` function for the main portion of the work.

5.7 rmanno

```
#include <tagUtils.h>

int rmanno(
        GapIO  *io,
        int     anno,
        int     lpos,
        int     rpos);
```

This function removes annotations in a specified region from an annotation list. The annotation list starts at annotation number *anno*. The new list head (which will change if we delete the first annotation) is returned. The region to remove annotations over is between base numbers *lpos* and *rpos* inclusive. Note that annotations overlapping this region, but not contained entirely within it, will have their either their position or length modified, or may need splitting in two. (Consider the case where a single tag spans the entire region to see where splitting is necessary.)

When succeeding the the new annotation number to form the annotation list head. Otherwise returns 0.

5.8 tag2values

```
#include <tagUtils.h>

int tag2values(
        char   *tag,
        char   *type,
        int    *start,
        int    *end,
        int    *strand,
        char   *comment);
```

This function converts a tag in string format to a tag represented by a series of separate integer/string values. It performs the opposite task to the `values2tag` function.

The tag string format is as used in the experiment file `TG` lines. The format is `"`*TYPE*<space>*S*<space>*start..end*`"` followed by zero or more comment lines, each starting with a newline character. *TYPE* is the tag type, which must be 4 characters, and *S* is the strand; one of `"+"`, `"-"` or `"b"` (both).

The tag string is passed as the *tag* argument. This is then expanded into the *type*, *start*, *end*, *strand* and *comment* values. The comment must have been allocated before hand (`strlen(tag)` will always be large enough). If no comment was found then *comment* is set to be an empty string. *type* should be allocated to be 5 bytes long.

The function returns 0 for success, -1 for failure.

5.9 values2tag

```
#include <tagUtils.h>

int values2tag(
        char   *tag,
        char   *type,
        int     start,
        int     end,
        int     strand,
        char   *comment);
```

This function converts a tag represented by a series of separate integer/string values to a single string of the format used by the experiment file `TG` line type. It performs the opposite task to the `tag2values` function.

For the format of the tag string please see _ref(tag2values, tag2values).

The *type*, *start*, *end*, *strand* and *comment* paramaters contain the current tag details. *comment* must be specified even when no comment exists, but can be specified as a blank string in this case. *tag* is expected to have been allocated already and no bounds checks are performed. A safe size for allocation is `strlen(comment)+30`.

The function returns 0 for success, -1 for failure.

5.10 rmanno_list

```
#include <tagUtils.h>

int rmanno_list(
        GapIO  *io,
        int     anno_ac,
        int    *anno_av);
```

This function removes a list of annotations from the database. The annotation lists for readings and contigs are also updated accordingly. The annotations numbers to remove are held in an array named *anno_av* with *anno_ac* elements.

This function returns 0 for success, -1 for failure.

5.11 insert_NEW_tag

```
#include <tagUtils.h>

void insert_NEW_tag(
        GapIO  *io,
        int     N,
        int     pos,
        int     length,
        char   *type,
        char   *comment,
        int     sense);
```

This function adds a new tag to the database. If *N* is positive, the tag is added to reading number *N*, otherwise it is added to contig number -*N*. The reading and contig annotation lists are updated accordingly.

The *pos*, *length*, *type*, *comment* and *sense* arguments specify the position, length, type (a 4 character string), comment and orientation of the tag to create. *comment* may be `NULL`. *sense* should be one of 0 for forward, 1 for reverse and 2 for both.

5.12 create_tag_for_gel

```
#include <tagUtils.h>

void create_tag_for_gel(
        GapIO  *io,
        int     gel,
        int     gellen,
        char   *tag);
```

This function is a textual analogue of the `insert_NEW_tag` function (which it uses). The function creates a new tag for a reading. The *gel* argument should contain the reading number and *gellen* the reading length. The tag to create is passed as the *tag* argument which is in the same format as taken by the *tag2values* function. See Section 5.8 [tag2values], page 93.

5.13 ctagget and vtagget

```
#include <tagUtils.h>

GAnnotations *ctagget(
        GapIO  *io,
        int     gel,
        char   *type);

GAnnotations *vtagget(
        GapIO  *io,
        int     gel,
        int     num_t,
        char  **type);
```

These function provides a mechanism of iterating around all the available tags of particular types on a given reading or contig number. The `ctagget` function searches for a single tag type, passed in *type* as a 4 byte string. The `vtagget` function searches for a set of tag types, passed as an array of *num_t* 4 byte strings.

To use the functions, call them with a non zero *gel* number and the tag type(s). The function will return a pointer to a *GAnnotations* structure containing the first tag on this reading or contig of this type. If none are found, `NULL` is returned.

To find the next tag on this reading or contig, of the same type, call the function with *gel* set to 0. To find all the tags of this type, keep repeating this until `NULL` is returned.

Returns a *GAnnotations* pointer for success, `NULL` for "not found", and `(GAnnotations *)-1` for failure. The annotation pointer returned is valid until the next call of the function.

For example, the following function prints information on all vector tags for a given reading.

```
void print_tags(GapIO *io, int rnum) {
    char *type[] = {"SVEC", "CVEC"};
    GAnnotations *a;

    a = vtagget(io, rnum, sizeof(types)/sizeof(*types), types);

    while (a && a != (GAnnotations *)-1) {
        printf("position %d, length %d\n",
            a->position, a->length);e

        a = vtagget(io, 0, sizeof(types)/sizeof(*types), types);
    }
}
```

5.14 tag_shift_for_insert

```
#include <tagUtils.h>

void tag_shift_for_insert(
        GapIO  *io,
        int     N,
        int     pos);
```

This function shifts or extends tags by a single base. The purpose is to handle cases where we need to insert into a sequence. An edit at position *pos* will mean moving every tag to the right of this one base rightwards. A tag that spans position *pos* will have it's length increased by one. If *N* is positive it specifies the reading number to operate on, otherwise it specifies the contig number (negated).

NOTE: This function **does not** work correctly for complemented readings. It is planned to fix this problem by creating a new function that operates in a more intelligent fashion. To work around this problem, logic similar to the following needs to be used.

```
/*
 * Adjust tags
 * NOTE: Must always traverse reading in reverse of original sense
 */
if (complemented) {
    for(i=j=0; i < gel_len; i++) {
        if (orig_seq[i] != padded_seq[j]) {
            tag_shift_for_insert(io, gel_num, length-j);
        } else
            j++;
    }
} else {
    for(i=j=gel_len-1; i >= 0; i--) {
        if (orig_seq[i] != padded_seq[j]) {
            tag_shift_for_insert(io, gel_num, j+1);
        } else
            j--;
    }
}
```

In the above example *padded_seq* is a padded copy of *orig_seq*. The function calls `tag_shift_for_insert` for each pad. Note that the order of the insertions is important and differs depending on whether the reading is complemented or not.

5.15 tag_shift_for_delete

```
#include <tagUtils.h>

void tag_shift_for_delete(
        GapIO  *io,
        int     N,
        int     pos);
```

This function shifts or shrinks tags by a single base. The purpose is to handle cases where we need to delete a base within a sequence. An deletion at position *pos* will mean moving every tag to the right of this position one base leftwards. A tag that spans position *pos* will have it's length decreased by one. If *N* is positive it specifies the reading number to operate on, otherwise it specifies the contig number (negated).

NOTE: This function **does not** work correctly for complemented readings. Also, it does not remove the tag when a deletion shrinks it's size to 0. It is planned to fix these problem by creating a new function that operates in a more intelligent fashion. To work around this problem, use logic similar to the example in `tag_shift_for_insert`. See Section 5.14 [tag_shift_for_insert], page 95.

5.16 type2str and str2type

```
#include <tagUtils.h>

int str2type(
        char   *stype);

void type2str(
        int     itype,
        char    stype[5]);
```

Note that these two functions are infact #defines. The prototypes are listed simply to guide their correct usage.

`str2type` converts a 4 character tag type, pointed to by *stype* into an integer value as used in the *GAnnotations.type* field.

`type2str` converts an integer type passed as *itype* to a 4 character (plus 1 nul) string.

6 Contig Registration Scheme

Each function wishing to access a contig on a long term basis needs to register itself before accessing the data. For example, the template display and contig editors should register, but show relationships produces a report taken from a single snap shot of the data and so does not need to register.

The idea of registering is to allow communication between views of the same (or derived) data, this insuring that they can be automatically kept up to date when modifications are made, and can provide mechanisms to prevent multiple, incompatible, edits of the same data. An example can be seen in the template display. Suppose we have the display showing contig 4. A join contig operation links this to contig 7 and produces a new contig — number 10. Other contig numbers may have shuffled too: if we had 9 contigs then contig 9 may well be renumbered to contig 4.

Therefore we need to notify any functions displaying contig 9 of a contig number change. We also need to notify displays of contig 4 that the number is now 10, and both this and contig 7 that the contents and length has changed.

Central to the scheme is the result manager. This displays a list of which data is registered and provides a further method for the user to interrogate specific results.

Notifications may also be for requesting data as well as informing changes. All registered items must respond to certain notifications, such as for determining the name of the function, so that it can be listed in the results manager.

6.1 Data Structures

For each contig we maintain a list of displays of this data. We register by supplying a function (of a specific type) to our registration scheme, along with any data of our own (called our *client_data*) that we wish to be passed back. When an operation is performed on this contig the function that we specified is called along with our own client_data and a description of the operation made. A function often does not need to be told of all changes, so when registering it's possible to list only those operations that should be responded to.

In addition to maintaining the above information, each registration contains an identifier, a time stamp, a type, and an `"id"` value.

The identifier is a simply number that is used to specify a single registered data, or a group of registered data. An example of it's use is within the contig selector; the selector is registered on all contigs, but each registration has the same identifier. A new identifier is returned by calling the `register_id` function.

The time stamp is allocated automatically when the `contig_register` function is called. It is displayed within the results manager.

The type is used to flag a registered data as belonging to a specific function. This is useful for when we wish to send a notification to all instances of a particular display, or to query whether the contig editor is running (such as performed by the stop codon display). The current types known are:

```
REG_TYPE_UNKNOWN
REG_TYPE_EDITOR
REG_TYPE_FIJ
REG_TYPE_READPAIR
REG_TYPE_REPEAT
REG_TYPE_QUALITY
REG_TYPE_TEMPLATE
REG_TYPE_RESTRICTION
REG_TYPE_STOPCODON
REG_TYPE_CONTIGSEL
REG_TYPE_CHECKASS
REG_TYPE_OLIGO
```

The id value is used to distinguish which pieces of data are connected. Each "result" has a single id value, but may consist of multiple pieces of registered data, all sharing the same id.

So the registration consists of the following structure:

```
typedef struct {
    void  (*func)(
                  GapIO    *io,
                  int       contig,
                  void     *fdata,
                  reg_data *jdata);
    void   *fdata;
    int     id;
    time_t  time;
    int     flags;
    int     type;
    int     uid; /* A _unique_ identifier for this contig_reg_t */
} contig_reg_t;
```

The `func` and `fdata` are the callback functions and client_data. *uid* is a number unique to all registrations, even those that have common *id* values. You need not be concerned about it's use; it is internal to the registration system.

Hence the total memory used by the registration system is an array of arrays of above structures. One array per contig, containing an array of *contig_reg_t* structs.

A notification of an action involves creating a *reg_data* structure and sending this to one of the notification functions (such as `contig_notify`). The *reg_data* structure is infact a union of many structure types; one for each notification type. In common to all these types is the job field. This must be filled out with the current notification type. See Section 6.4 [The Notifications Available], page 103.

As *reg_data* is a union of structures, it must be access by a further pointer indirection. For instance, to determine the position of the contig editor cursor from a `REG_CURSOR_NOTIFY` notification we need to write `"reg_data->cursor_notify->pos"` rather than simply `"reg_data->pos"`. The complete list of union names can be found in io-reg.h. The current list is summarised below. The types and use of these structures will be discussed in further detail later.

```
typedef union _reg_data {
    /* MUST be first here and in job data structs */
    int job;
    reg_generic         generic;
    reg_number          number;
    reg_join            join;
    reg_order           order;
    reg_length          length;
    reg_query_name      name;
    reg_delete          delete;
    reg_complement      complement;
    reg_get_lock        glock;
    reg_set_lock        slock;
    reg_quit            quit;
    reg_get_ops         get_ops;
    reg_invoke_op       invoke_op;
    reg_params          params;
    reg_cursor_notify   cursor_notify;
    reg_anno            annotations;
    reg_register        c_register;
    reg_deregister      c_deregister;     reg_highlight_read  highlight;
    reg_buffer_start    buffer_start;
    reg_buffer_end      buffer_end;
} reg_data;
```

6.2 Registering a Piece of Data

To register data several things need to be known; the contig number, the callback function, the client_data (typically the address of the data to register), the list of notifications to respond to, an indentifier, and the "type" of this data (one of the `REG_TYPE_` macros).

If the data needs updating when more than one specific contig changes, then the data should be registered with more than one contig.

Use the `contig_register` function to register an item. The prototype is:

```
#include <io-reg.h>

int contig_register(
        GapIO  *io,
        int     contig,
        void  (*func)(
                      GapIO     *io,
                      int        contig,
                      void      *fdata,
                      reg_data  *jdata),
        void   *fdata,
        int     id,
        int     flags,
        int     type);
```

contig is a contig number in the C sense (`1` to `NumContigs(io)`), not a gel reading number.

The *fdata* (the client_data mentioned before) can be anything you wish. It will be passed back to the callback function *func* when a notification is made. Typically it's best to simply pass the address of your data that you wish to keep up to date. If your data is not a single pointer then turn it into one by creating a structure containing all the relevant pointers.

The id number is usually unique for each time an option it ran, but common to all registrations of this particular piece of data. This is not a hard and fast rule — it depends on how you wish to interact with this data. For instance, the contig selector window registers with all contigs so that it can be notified when any contig changes. The same *id* is used for each of these registrations as it is the collection of registrations as a whole which is required for the display.

"Flags" is used to request which notifications should be sent to this callback function. Each notification has a name which is actually a #define for a number. This names can be ORed together to generate a bit field of acknowledged requests. There are some predefined bitfields (for shortening the function call) that can themselves be ORed together. See Section 6.4 [The Notifications Available], page 103. Finally, one special flag can be ORed on to request that this function does not appear in the results manager window. This flag is `REG_FLAG_INVIS`: see the contig selector code for an example.

An example of using `contig_register` can be seen in the stop codon plot. Our stop codon results are all held within a structure of type *mobj_stop*. The general outline of our stop codon code is as follows:

```
mobj_stop *s;
int id;

if (NULL == (s = (mobj_stop *)xmalloc(sizeof(mobj_stop)))) {
    return 0;
}

[ Fill in our 's' structure with our results ]

DrawStopCodons(s);
id = register_id();
contig_register(io, contig_number, stop_codon_callback, (void *)s, id,
                REG_REQUIRED | REG_DATA_CHANGE | REG_OPS | REG_GENERIC
                | REG_NUMBER_CHANGE | REG_REGISTERS | REG_CURSOR_NOTIFY,
                REG_TYPE_STOPCODON);
```

Here we've requested that the result *s*, of type `REG_TYPE_STOPCODON`, should be passed to the `stop_codon_callback` function whenever a notification of type `REG_REQUIRED`, `REG_DATA_CHANGE`, `REG_OPS`, `REG_GENERIC`, `REG_NUMBER_CHANGE`, `REG_REGISTERS` or `REG_CURSOR_NOTIFY` occurs. These notification types are actually combinations of types, but more on this later.

6.3 The Callback Function

The callback function must be of the following prototype:

```
void function(
        GapIO     *io,
        int        contig,
        void      *fdata,
        reg_data  *jdata);
```

Here *fdata* will be the client_data specified when registering. The first task within our callback function will be to cast this to a useful type. As the type of this *fdata* will change depending on what piece of data is registered this is a required, but tedious, action.

The next task at hand is to see exactly why the callback function was called. This is listed in the *reg_data* parameter. Specifically `jdata->job` will be one of the many notification types. The suggested coding method is to perform a switch on this field as follows:

```
static void some_callback(GapIO *io, int contig, void *fdata, reg_data *jdata)
{
    some_type_t *s = (some_type_t *)fdata;

    switch(jdata->job) {
    case REG_QUERY_NAME:
        sprintf(jdata->name.line, "Some name");
        break;

    case REG_QUIT:
    case REG_DELETE:
        ShutDownSomeDisplay(fdata);
        xfree(fdata);
        break;
    }
}
```

REG_QUERY_NAME, REG_QUIT, REG_DELETE and REG_PARAMS are required to be accepted by all registered items.

In general the callback function will also be interested in changes to the contig that the data is registered with. These involve the REG_JOIN_TO, REG_COMPLEMENT, REG_LENGTH, REG_NUMBER_CHANGE and REG_ANNO requests.

For precise details on handling the various notifications, please see the following section.

6.4 The Notifications Available

In order to shorten code, especially when requesting which notifications should be accepted using the contig_register call, the following macros may be of use. They are used to group the various notifications.

```
#define REG_REQUIRED    (REG_QUERY_NAME | REG_DELETE | REG_QUIT | REG_PARAMS)
#define REG_DATA_CHANGE (REG_JOIN_TO | REG_LENGTH | REG_COMPLEMENT)
#define REG_OPS         (REG_GET_OPS | REG_INVOKE_OP)
#define REG_LOCKS       (REG_GET_LOCK | REG_SET_LOCK)
#define REG_REGISTERS   (REG_REGISTER | REG_DEREGISTER)
#define REG_BUFFER      (REG_BUFFER_START | REG_BUFFER_END)
#define REG_ALL         (REG_REQUIRED | REG_DATA_CHANGE | REG_OPS | REG_LOCKS\
                         | REG_ORDER | REG_CURSOR_NOTIFY | REG_NUMBER_CHANGE \
                         | REG_ANNO | REG_REGISTERS | REG_HIGHLIGHT_READ \
                         | REG_BUFFER)
```

In the following descriptions, we outline the different notifications in the format of name followed by the name within the *reg_data* structure, the structure itself, and the description.

6.4.1 REG_GENERIC

```
reg_generic         generic;

typedef struct {
    int    job;        /* REG_GENERIC */
    int    task;       /* Some specific task */
    void  *data;      /* And data associated with the task */
} reg_generic;
```

This is used for sending specific requests to specific data or data types. The task is a macro named after the type the task deals with. Eg TASK_EDITOR_SETCURSOR. REG_GENERIC is usually used in conjuction with a result_notify or type_contig_notify function call. See Section 6.6 [Specific Notification Tasks], page 111.

6.4.2 REG_NUMBER_CHANGE

```
reg_number          number;

typedef struct {
    int    job;        /* REG_NUMBER_CHANGE */
    int    number;     /* New contig number */
} reg_number;
```

Sent whenever a contig number changes, but not when a reading number changes. This is currently only sent when renumbering contigs during a contig delete operation.

6.4.3 REG_JOIN_TO

```
reg_join            join;

typedef struct {
    int    job;        /* REG_JOIN_TO */
    int    contig;     /* New contig number */
    int    offset;     /* Offset of old contig into new contig */
} reg_join;
```

Used to notify data that this contig has just been joined to another contig, at a specified offset. *contig* is contig number that this contig has been joined to (and hence it's new number). *offset* is the offset within the new contig that the old contig has been joined to. This request is always sent to the right most of the contig pair to join. The leftmost contig receives a REG_LENGTH notification. See Section 6.9.2 [Joining Two Contigs], page 117.

6.4.4 REG_ORDER

```
reg_order           order;

typedef struct {
    int    job;        /* REG_ORDER */
    int    pos;        /* New order */
} reg_order;
```

The purpose is to inform when the contig order changes. *pos* is the new position of this contig. To be consistent, there will be further *REG_ORDER* requests indicating the new position of the contig that was previously at this position. Typically this is simply handled by sending a notification for each contig. To handle these efficiently it is probably best to use the REG_BUFFER_START and REG_BUFFER_END notifications.

6.4.5 REG_LENGTH

```
reg_length          length;

typedef struct {
    int    job;        /* REG_LENGTH, implies data change too */
    int    length;     /* New length */
} reg_length;
```

Sent whenever the length or data within of a contig changes. In this respect `REG_LENGTH` is a bit of a misnomer; replacing a single base within the contig editor and then saving (which does not change the length of that contig) will still send a `REG_LENGTH` request to inform data that the contig has changed. This is one of the most frequently sent and acknowledged requests.

6.4.6 REG_QUERY_NAME

```
reg_query_name      name;

typedef struct {
    int    job;        /* REG_QUERY_NAME */
    char  *line;     /* char[80] */
} reg_query_name;
```

Sent by the `result_names` routine to obtain a brief one line (less than 80 characters) name of this registered item. Callback procedures should write into the *line* field themselves with no need for memory allocation. The name returned here will be used as a component of the line within the Results Manager window. Registered data is required to handle this request, unless it is invisible (has the `REG_FLAG_INVIS` bit set).

6.4.7 REG_DELETE

```
reg_delete          delete;

typedef struct {
    int    job;        /* REG_DELETE */
} reg_delete;
```

The registered data should be removed and any associated displays should be shutdown. This is in response to a contig being deleted (by the `io_delete_contig` function), or a programmed shutdown to force associated displays to quit (such as when forcing the quality display to quit when the user quits the template display). Registered data is required to handle this request.

6.4.8 REG_GET_LOCK and REG_SET_LOCK

```
#define REG_LOCK_READ   1
#define REG_LOCK_WRITE  2

reg_get_lock        glock;
reg_set_lock        slock;

typedef struct {
    int    job;        /* REG_GET_LOCK */
    int    lock;       /* Sends lock requirements, returns locks allowed */
} reg_get_lock, reg_set_lock;
```

Both these notifications share the same structure. The pair are used in conjunction to determine whether exclusive write access is allowed on this contig, and if so to set this access. This is all managed by the `contig_lock_write` function. See Section 6.8 [Locking Mechanisms], page 115. Functions wishing to modify data, such as complement, should use locking.

6.4.9 REG_COMPLEMENT

```
reg_complement      complement;

typedef struct {
    int    job;        /* REG_COMPLEMENT */
} reg_complement;
```

Notifies that the contig has just been complemented. It may prove easy to simply handle this and other data change notifications all the same. However in slow functions, it may be quicker to handle complement functions separately, as it can be quicker to complement result data than to recalculate it.

6.4.10 REG_PARAMS

```
reg_params          params;

typedef struct {
    int    job;        /* REG_PARAMS */
    char  *string;     /* Pointer to params string */
} reg_params;
```

Sent as a request for obtaining the parameters used for generating this data. Note that in contrast to `REG_NAME` the *string* field here is not already allocated. The function acknowledging this request should point *string* to a static buffer of it's own. Currently, although implemented, this request is not used.

6.4.11 REG_QUIT

```
reg_quit            quit;

typedef struct {
    int    job;        /* REG_GET_LOCK */
    int    lock;       /* Sends lock requirements, returns locks allowed */
} reg_quit;
```

Sent to request a shutdown for this display. This is not like `REG_DELETE`n, whereby the data is told that it must shutdown as the contig has already been deleted. If a display cannot shutdown (for example it is a contig editor that has unsaved data) the lock should be cleared and the calling function should check this to determine whether the shutdown succeeded. This is handled internally by the `tcl_quit_displays` function.

6.4.12 REG_CURSOR_NOTIFY

```
reg_cursor_notify   cursor_notify;

typedef struct {
    int    job;           /* REG_CURSOR_NOTIFY */
    int    editor_id;     /* Which contig editor */
    int    seq;           /* Gel reading number (0 == consensus) */
    int    pos;           /* Position in gel reading */
} reg_cursor_notify;
```

Sent by the contig editor at startup and whenever the editing cursor moves. The *editor_id* is a number unique to each contig editor, so it is possible to distinguish different editors. *seq* is either 0 for the consensus, or a gel reading number. *pos* is the offset within that gel reading, rather than the total offset into the consensus (unless *seq* is 0).

6.4.13 REG_GET_OPS

```
reg_get_ops         get_ops;

typedef struct {
    int    job;      /* REG_GET_OPS */
    char  *ops;      /* Somewhere to place ops in, unalloced to start with */
} reg_get_ops;
```

Within the Results Manager a popup menu is available for choosing from a list of tasks to be performed on this data. These can include anything, but typically include deleting the data and listing textual information. The *ops* field will intitially point to `NULL` when the callback function is called. The callback function should then assign ops to a static string listing `NULL` separated items to appear on the popup menu, ending in a double `NULL`. If an item in this string is `"SEPARATOR"`, a separator line on the menu will appear. If an item is `"PLACEHOLDER"`, then nothing for this item will appear in the menu, but the numbering used for `REG_INVOKE_OP` will count `"PLACEHOLDER"` as an option. An example of the acknowledging code follows:

```
case REG_GET_OPS:
    if (r->all_hidden)
        jdata->get_ops.ops = "Information\0PLACEHOLDER\0"
            "Hide all\0Reveal all\0SEPARATOR\0Remove\0";
    else
        jdata->get_ops.ops = "Information\0Configure\0"
            "Hide all\0Reveal all\0SEPARATOR\0Remove\0";
    break;
```

Here we have a menu containing, "Information", "Configure", "Hide all", "Reveal all" and "Remove". In this example, if `r->all_hidden` is set then the "Configure" option does not appear, but the later options (eg Remove) will always be given the same number (4 in this case).

6.4.14 REG_INVOKE_OP

```
reg_invoke_op       invoke_op;

typedef struct {
    int    job;        /* REG_INVOKE_OP */
    int    op;         /* Operation to perform */
} reg_invoke_op;
```

When the user has chosen an option from the Results Manager popup window (from the list returned by `REG_GET_OPS`), `REG_INVOKE_OP` is called with an integer value (held in the *op* field) detailing which operation was chosen. *op* starts counting from zero for the first item returned from `REG_GET_OPS`, and counts up one each time for each operation or `PLACEHOLDER` listed. An example of an acknowledge for `REG_INVOKE_OP` to complement the example given in `REG_GET_OPS` follows:

```
case REG_INVOKE_OP:
    switch (jdata->invoke_op.op) {
    case 0: /* Information */
        csmatch_info((mobj_repeat *)r, "Find Repeats");
        break;
    case 1: /* Configure */
        csmatch_configure(io, cs->window, (mobj_repeat *)r);
        break;
    case 2: /* Hide all */
        csmatch_hide(our_interp, cs->window, (mobj_repeat *)r, csplot_hash);
        break;
    case 3: /* Reveal all */
        csmatch_reveal(our_interp, cs->window, (mobj_repeat *)r, csplot_hash);
        break;
    case 4: /* Remove */
        csmatch_remove(io, cs->window, (mobj_repeat *)r, csplot_hash);
        break;
    }
    break;
```

6.4.15 REG_ANNO

```
    reg_anno            annotations;

    typedef struct {
        int    job;        /* REG_ANNO */
    } reg_anno;
```

Sent when only the annotations (tags) for a contig have been updated. It is sometimes simplest for clients to handle `REG_ANNO` in the same manner as `REG_LENGTH`. However in some cases it can be much more efficient to handle separately as it may be easier to redisplay annotations than to redisplay everything.

6.4.16 REG_REGISTER and REG_DEREGISTER

```
    reg_register        c_register;
    reg_deregister      c_deregister;

    typedef struct {
        int    job;        /* REG_REGISTER, REG_DEREGISTER */
        int    id;         /* Registration id */
        int    type;       /* Registration type */
        int    contig;     /* Contig number */
    } reg_register, reg_deregister;
```

Both of these notifications share the same structure. They are sent whenever a registration or deregistration of another piece of data is performed for this contig. An example of the use of this is within the stop codon display which enables use of the "Refresh" button when a contig editor is running. The *id*, *type* and *contig* fields here are the same as the fields with the same name from the *contig_reg_t* structure.

6.4.17 REG_HIGHLIGHT_READ

```
    reg_highlight_read  highlight;

    typedef struct {
        int    job;       /* REG_HIGHLIGHT_READ */
        int    seq;       /* Gel reading number (-ve == contig consensus) */
        int    val;       /* 1==highlight, 0==dehighlight */
    } reg_highlight_read;
```

This is used for notifying that an individual reading has been highlighted. It's purpose is to allow displays to synchronise highlighting of data. For instance, both the contig editor and template display send and acknowledge this notification. Thus when a name in the editor is highlighted the template display will highlight the appropriate reading, and vice versa.

When *seq* is positive it represents the reading to highlight, otherwise it is 0 minus the contig number (not leftmost reading number).

6.4.18 REG_BUFFER_START and REG_BUFFER_END

```
reg_buffer_start    buffer_start;
reg_buffer_end      buffer_end;

typedef struct {
    int    job;
} reg_buffer_start, reg_buffer_end;
```

These two notifications share the same structure, which holds no information. The purpose of `REG_BUFFER_START` is simply as a signal that many notifications will be arriving in quick succession, until a `REG_BUFFER_END` request arrives. The purpose is to speed up redisplay of functions registered with many contigs.

As an example consider the enter tags function. This adds tags to many, potentially all, contigs. We can keep track of which contigs we need to send `REG_ANNO` requests to, and send them with code similar to the following:

```
/* Notify of the start of the flurry of updates */
rs.job = REG_BUFFER_START;
for (i = 0; i < NumContigs(args.io); i++) {
    if (contigs[i]&1) {
        contig_notify(args.io, i+1, (reg_data *)&rs);
    }
}

/* Now notify all the contigs that we've added tags to */
ra.job = REG_ANNO;
for (i = 0; i < NumContigs(args.io); i++) {
    if (contigs[i]&1) {
        contig_notify(args.io, i+1, (reg_data *)&ra);
    }
}

/* Notify of the end of the flurry of updates */
re.job = REG_BUFFER_END;
for (i = 0; i < NumContigs(args.io); i++) {
    if (contigs[i]&1) {
        contig_notify(args.io, i+1, (reg_data *)&re);
    }
}
```

Consider the action of the contig selector. This needs to refresh the display whenever any modifications are made, including annotations. The enter tags function needs to send notifications to many contigs, thus the contig selector will receive many requests. It is obviously more efficient for the contig selector to only redisplay once. The addition of `BUFFER_START` and `BUFFER_END` solve this. As we don't know exactly which functions will be registered with which contigs, the enter tags code has to notify every contig. Hence the contig selector code must keep a count on the start and end of buffers so that it only needs to redisplay on the last buffer end. This code is as follows (tidied up and much shortened for brevity):

```
switch(jdata->job) {
case REG_BUFFER_START:
    {
        cs->buffer_count++;
        cs->do_update = REG_BUFFER_START;
        return;
    }

case REG_BUFFER_END:
    {
        cs->buffer_count--;
        if (cs->buffer_count <= 0) {
            cs->buffer_count = 0;
            if (cs->do_update & REG_LENGTH) {
                [ Redisplay Contigs ]
            } else if (cs->do_update & REG_ANNO) {
                [ Redisplay Tags ]
            } else if (cs->do_update & REG_ORDER) {
                [ Shuffle Order]
            }
            cs->do_update = 0;
        }
        return;
    }

case REG_ANNO:
    {
        if (!cs->do_update) {
            [ Redisplay Tags ]
        } else {
            cs->do_update |= REG_ANNO;
        }
        return;
    }
/* etc */
```

For further examples of handling buffering see the template display code.

6.5 Sending a Notification

When a function modifies data it is the responsibility of this function to inform others, via the contig registration scheme, of this change. At the time of notification the data on disk and in memory should be consistent (ie that check_database should not fail). To illustrate this, when joining two contigs we should not start sending notifications until we've recomputed the lengths and left/right neighbours of the joined contig.

To send a request, one of the notification functions should be used. The simplest of these is `contig_notify`. This function takes a *GapIO* pointer, a contig number, and a *reg_data* pointer as arguments. The *reg_data* is the union of notification types outlined in the above sections. The separate steps for notifying are:

1. Create a variable of the appropriate structure type (eg `reg_length`).
2. Fill the job field of this structure with the correct definition (eg `REG_LENGTH`).
3. Fill in any structure dependant fields of the structure (eg *length* in the case of `reg_length`).
4. Call `contig_notify` with the *GapIO*, contig number and notification structure. The notification structure should be cast back to a pointer to the *reg_data* union type.

An example illustrating the above steps would be:

```
reg_length jl;

[...]

jl.job = REG_LENGTH;
jl.length = some_length;
contig_notify(io, contig_number, (reg_data *)&jl);
```

The available notification functions are `contig_notify`, `result_notify`, `type_notify` and `type_contig_notify`. See Section 6.7 [C Functions Available], page 112.

6.6 Specific Notification Tasks

Some registered items may support extra forms of communication than the listed notifications. In this case, we use the `REG_GENERIC` notification together with a task number and some task specific data to send a specific task to a specific registered data. This provides a way for individual displays to add new communicates methods to the registration scheme.

To send a `REG_GENERIC` task, the *reg_generic* structure must first be completed by setting *job*, *task* and *data*. *Data* will point to another structure, which is unique for specific type of task. The task data structure must then be initialised and sent to the appropriate client contig, id or type.

The *task* number needs to be unique across all the types of generic tasks likely to be sent to the client. For instance, a contig editor can receive `TASK_EDITOR_SETCURSOR` and `TASK_EDITOR_GETCON` tasks. Obviously the `#define`s for these tasks need to be different. However they may safely coincide with `TASK_TEMPLATE_REDRAW`, which is used by the template display, as we know that the the editor will never receive this task (and vice versa). The assignment of task numbers is at present something which requires further investigation. However the use of defines everywhere means that they are trivial to change.

6.6.1 TASK_EDITOR_GETCON

```
typedef struct {
    char  *con;         /* Allocated by the contig editor */
    int    lreg;        /* Set lreg and rreg to 0 for all consensus */
    int    rreg;
    int    con_cut;
    int    qual_cut;
} task_editor_getcon;
```

Allocates and calculates a consensus (stored in *con*) between *lreg* and *rreg*. If *lreg* and *rreg* are both zero, then all the consensus is computed. The calling function is expected to free *con* when finished. An example of use can be seen in the stop codon code:

```
reg_generic gen;
task_editor_getcon tc;
```

```
    gen.job = REG_GENERIC;
    gen.task = TASK_EDITOR_GETCON;
    gen.data = (void *)&tc;

    tc.lreg = 0;
    tc.rreg = 0;
    tc.con_cut = consensus_cutoff;
    tc.qual_cut = quality_cutoff;

    if (type_contig_notify(args.io, args.contig, REG_TYPE_EDITOR,
                           (reg_data *)&gen, 0) == -1)
        return TCL_OK;

    [...]

    xfree(tc.con);
```

6.6.2 TASK_CANVAS_SCROLLX

6.6.3 TASK_CANVAS_SCROLLY

6.6.4 TASK_CANVAS_ZOOMBACK

6.6.5 TASK_CANVAS_ZOOM

6.6.6 TASK_CANVAS_CURSOR_X

6.6.7 TASK_CANVAS_CURSOR_Y

6.6.8 TASK_CANVAS_CURSOR_DELETE

6.6.9 TASK_CANVAS_RESIZE

6.6.10 TASK_CANVAS_REDRAW

6.6.11 TASK_CANVAS_WORLD

6.6.12 TASK_WINDOW_ADD

6.6.13 TASK_WINDOW_DELETE

6.6.14 TASK_CS_REDRAW

6.6.15 TASK_RENZ_INFO

6.6.16 TASK_TEMPLATE_REDRAW

6.6.17 TASK_DISPLAY_RULER

6.6.18 TASK_DISPLAY_TICKS

6.7 C Functions Available

The prototypes for all of these functions can be found in '`io-reg.h`'. The code for these functions is held in '`io-reg.c`'.

6.7.1 contig_register_init

```
#include <io-reg.h>

int contig_register_init(GapIO  *io);
```

Initialises the contig register lists. This is only performed once, upon opening of a new database. The registration lists are automatically extended when new contigs are created.

The function returns 0 for succes, -1 for error.

6.7.2 register_id

```
int register_id();

Returns: the id (always a non zero value).
```

Returns a new id number for use as the id field to be sent to a `contig_register` call. Each time this function is called a new number is returned.

6.7.3 contig_register

```
int contig_register(GapIO *io, int contig,
                    void (*func)(GapIO *io, int contig, void *fdata,
                                 reg_data *jdata),
                    void *fdata,
                    int id, int flags, int type);
Returns:  0 for success
         -1 for error.
```

Registers "func(io, contig, fdata, jdata)" with the specified contig. This doesn't check whether the (func,fdata) pair already exist for this contig.

6.7.4 contig_deregister

```
int contig_deregister(GapIO *io, int contig,
                      void (*func)(GapIO *io, int contig, void *fdata,
                                   reg_data *jdata),
                      void *fdata);

Returns:  0 for success
         -1 for error.
```

Deregisters "func(io, contig, fdata, jdata)" from the specified contig. The (func,fdata) pair must match exactly to deregister.

6.7.5 contig_notify

```
void contig_notify(GapIO *io, int contig, reg_data *jdata);
```

Sends a notification request to all items registered with the specified contig.

6.7.6 contig_register_join

```
int contig_register_join(GapIO *io, int cfrom, int cto);

Returns:  0 for success
         -1 for error.
```

Joins two registration lists. This adds all items listed on the registration list for contig 'cfrom' to the registration list for contig 'cto'. Entries that are registered on both lists are not duplicated. The 'cfrom' registration list is left intact.

6.7.7 result_to_regs

```
contig_reg_t **result_to_regs(GapIO *io, int id);

Returns:  An allocated list of contig_reg_t pointers upon success.
          NULL for failure.
```

Converts an id number to an array of *contig_reg_t* pointers. The *contig_reg_t* structures pointed to are considered the property of the registration scheme and should not be modified. The caller is expect to deallocate the returned list by calling the `xfree` function.

6.7.8 result_names

```
char *result_names(GapIO *io, int *contig, int *reg, int *id, int first);

Returns: The next name upon success.
         NULL for failure.
```

Generates description of functions registered with a particular contig. If contig 0 is specified then all are listed. 'contig' is modified to return the contig number this result was from (useful when sending contig 0), as is 'reg' to return the index into the registration array for this contig. This (contig,reg) pair specifies a particular result without the need for remembering pointers. 'id' contains a unique id number for this result.

6.7.9 result_time

```
char *result_time(GapIO *io, int contig, int id);

Returns: The time for success.
         "unknown" for failure.
```

Given a specific contig and id number, returns a string describing the time a specific id was registered. This assumes that all registered items with this id was registered at the same time. The string is statically allocated and should be be freed.

6.7.10 result_notify

```
void result_notify(GapIO *io, int id, reg_data *jdata, int all);
```

Sends a notification request to registered data with the specified id. If 'all' is non zero then all registered data with this id will be notified, otherwise only the first instance of this id found will be notified.

6.7.11 result_data

```
void *result_data(GapIO *io, int id, int contig);

Returns:  contig_reg_t->data for id upon success
          NULL upon failure.
```

Returns the data component of a *contig_reg_t* structure for a specific id. If id represents more than one piece of data, the first found (the search order is undefined) is returned. If the contig is specified then id will be search for only within this contig registration list, otherwise (when contig is zero) all contigs are scanned.

6.7.12 type_to_result

```
int type_to_result(GapIO *io, int type, int contig);

Returns:  id value for success.
          0 for failure.
```

Returns the first id value found for a given id. If contig is specifed as a non zero value we search for id only within this contig. Otherwise all contigs are scanned.

6.7.13 type_notify

```
int type_notify(GapIO *io, int type, reg_data *jdata, int all);

Returns:  0 for success
         -1 when none of this type were found.
```

Sends a notification request to registered data with the specified type. If 'all' is non zero then all registered data with this type will be notified, otherwise only the first instance of this type found will be notified.

6.7.14 type_contig_notify

```
int type_contig_notify(GapIO *io, int contig, int type,
                       reg_data *jdata, int all);

Returns:  0 for success
         -1 when none of this type were found.
```

Sends a notification request to registered data of a given type only within the specified contig. If 'all' is non zero then all registered data with this type in this contig will be notified, otherwise only the first instance of this type found will be notified.

6.8 Locking Mechanisms

When preparing to update data it is essential that a function checks whether other displays are currently accessing this data, and if so whether these displays are allowing the data to be modified.

This is implemented with use of the REG_GET_LOCK and REG_SET_LOCK notifications. These notifications both both include a lock field within their structures. This is initially set to the mode of access desired (currently REG_LOCK_WRITE is the only one we support). The `contig_notify` call is then used to send this notification to all appropriate data callbacks. If a callback wishes to block the request to write it should clear this lock flag.

The calling code then checks the returned status of the lock flag. If the REG_LOCK_WRITE bit is still set then it knows locking is allowed. In this case notification of the acceptance of this lock is sent around using the REG_SET_LOCK request. An example of the communication follows. To send the lock request we do:

```
reg_get_lock lg;

lg.job = REG_GET_LOCK;
lg.lock = REG_LOCK_WRITE;

contig_notify(io, contig, (reg_data *)&lg);
```

The default action of ignoring the REG_GET_LOCK request will allow the write operation to take place. The contig editor does not support updates of the contig that it is editing other than those made by itself, so it needs to block such locks. The callback procedure of the contig editor contains:

```
        case REG_GET_LOCK:
            /*
             * We need exclusive access, so clear any write lock
             */
            if (jdata->glock.lock & REG_LOCK_WRITE)
                jdata->glock.lock &= ~REG_LOCK_WRITE;

            break;
```

The calling code should now check the status of the lock and send a REG_SET_LOCK request if the lock was not blocked:

```
        if (lg.lock & REG_LOCK_WRITE) {
            reg_set_lock ls;

            ls.job = REG_SET_LOCK;
            ls.lock = REG_LOCK_WRITE;

            contig_notify(io, contig, (reg_data *)&ls);

            [ ... ]
        }
```

To simplify this procedure, the `contig_lock_write` function performs the above lock request and acknowledge protocol.

```
    int contig_lock_write(GapIO *io, int contig);

    Returns:  0 for success (write granted)
             -1 for failure (write blocked)
```

In some cases, where large amounts of data are modified in unpredictable fashion, it is easier to simply shut down all displays viewing the database before proceding. This is especially true of functions such as assembly where all contigs maybe modified. In this case we use the locking mechanism once more, except with a REG_QUIT call instead of REG_GET_LOCK. The same procedure of checking and clearing (if necessary) the lock flag is used. Once again, an example from the contig editor callback illustrates the procedure.

```
        case REG_QUIT:
            /*
             * We are being asked to quit. We can only allow this is we
             * haven't made changes.
             */
            if (_editsMade(db)) {
                jdata->glock.lock &= ~REG_LOCK_WRITE;
            } else {
                DBI_callback(db, DBCALL_QUIT, 0, 0, NULL);
            }

            break;
```

The code above checks whether the editor has made any edits. If not the editor is shutdown, otherwise the REG_LOCK_WRITE flag is cleared.

The `tcl_quit_displays` function can be used to perform the REG_QUIT locking procedure. Currently this is an interface to Tcl and no C interface, other than using the contig_notify with REG_QUIT, exists. See Section 6.10 [Tcl Interfaces], page 118.

6.9 Examples of Specific Functions

Here we describe in detail how certain operations interact with the contig registration. They are described here because the notifications generated may not be immediately obvious.

6.9.1 Deleting a contig

As contig numbers must always be from 1 to N, where N is the number of contigs, if we remove a particular contig, we need to ensure we still have contigs 1 to N-1. In thise case, deleting contig x, where x != N, will mean that we have a hole (at x) which can be filled by moving N down to x.

To illustrate in an algorithm we have the following; Given N contigs and a request to delete contig x.

1. Delete contig x. This is a NULL operation as far as the `io_delete_contig` operation goes as we're already assuming the data on this contig has gone elsewhere.
2. Move contig N to contig x (if x != N). This includes updating the disk images as well as the fortran arrays and the contig order, but not the registration lists — yet.
3. Decrement the number of contigs. (N–)
4. Notify contig x of the delete using REG_DELETE.
5. Notify contig N of the renumber to contig x using REG_NUMBER_CHANGE. (if appropriate)
6. Update registration list information.

Hence it is important to remember that after an `io_delete_contig` the contig numbers may not be the same as before the call.

6.9.2 Joining two contigs

The order of events within the joining is crucial. In the past several bugs have arisen due to this order being incorrect. We need to notify both the left and right contigs of the change, to join the two registration lists, and to delete the contig. Deleting the contig must be the last operation as this may renumber one of our contigs.

The order used is as follows, assuming we are joining two contigs together. We join 'left' to 'right', giving a new contig 'left'.

1. Perform the actual join of the data. This involves updating everything except without notifications and without modifying the registration lists.
2. Send a REG_JOIN_TO request to 'right' informing the new contig number is 'left'. This also includes the offset of 'right' within 'left'.
3. Merge the registration lists using `contig_register_join`. We copy 'right' to 'left', leaving 'right' unchanged. It is required to leave 'right' unchanged so that the delete request is acknowledged.
4. Notify 'left' of a change of length using REG_LENGTH. Note that this now also includes notifying items previously register with 'right'.
5. Delete contig 'right'. As shown above, this will generate REG_DELETE and possibly REG_NUMBER_CHANGE requests.

6.10 Tcl Interfaces

Some of the contig registration scheme needs to be visible at the Tcl/Tk level. This includes, amongst other things, anything to do with the Results Manager window. The complete list of Tcl callable functions can be found in tk-io-reg.h. The functions are described below.

6.10.1 clear_cp

```
clear_cp -io handle -id number

Returns: nothing
```

This command removes (sends a `REG_QUIT` request) all registered items that have displays on the contig comparator window. Currently this list is hard coded to include the following types: `REG_TYPE_FIJ`, `REG_TYPE_READPAIR`, `REG_TYPE_REPEAT`, `REG_TYPE_CHECKASS`, `REG_TYPE_OLIGO`.

The contig comparator is then turned back into the 1D contig selector window. The *id* of the contig comparator is needed for this.

6.10.2 clear_template

```
clear_template -io handle -id number

Returns: nothing
```

This command deletes all items on the template display with an id of *number*. It loops through all windows contained within this template display, sending a `REG_QUIT` request to them.

FIXME: This doesn't appear to remove either the template display itself or the ruler. Is it meant to?

6.10.3 register_id

```
register_id

Returns: the id.
```

A Tcl interface to the `register_id` function.

6.10.4 result_names

```
result_names -io handle

Returns: a list describing all results.
```

A Tcl interface to the `result_names` function. This produces a single string describing the complete list of results. The format is "{contig regnum id string} ?{contig regnum id string}? ..." and so can be accessed as a Tcl list.

6.10.5 result_time

```
result_time -io handle -contig contig_number -id id_number

Returns: the time in string format.
```

A Tcl interface to the `result_time` function.

6.10.6 result_delete

```
result_delete -io handle -id id_number

Returns: nothing
```

Sends a REG_DELETE request to a specific id.

6.10.7 result_quit

```
result_quit -io handle -id id_number

Returns: nothing
```

Sends a REG_QUIT request to a specific id.

6.10.8 reg_get_ops

```
reg_get_ops -io handle -id id_number

Returns: a Tcl list of available operations.
```

A Tcl interface to the REG_GET_OPS notification.

6.10.9 reg_invoke_op

```
reg_invoke_op -io handle -id id_number -option option_number

Returns: nothing
```

A Tcl interface to the REG_INVOKE_OP notification.

6.10.10 reg_notify_update

```
reg_notify_update -io handle -contig contig_number

Returns: nothing
```

Sends a REG_LENGTH request to a specific contig, or to all contigs if contig_number is specified as 0.

6.10.11 reg_notify_highlight

```
reg_notify_highlight -io handle -reading identifier -highlight value

Returns: nothing
```

Sends a REG_HIGHLIGHT request to a specific contig, indicating that the highlight value of the specified *reading_number* is *value*. The reading is specified as an *identifier* consisting of the name, #reading_number or =contig_number.

6.10.12 quit_displays

```
quit_displays io_handle function_name

Returns:  0 for success
         -1 for failure
```

Sends a REG_QUIT request to all registered data. If an error occurs, a database busy message is sent to the error window with the "function_name" listed.

6.11 Future Enhancements

1. Rationalise naming and arguments to functions:
   1. Some Tcl interfaces don't take `"-io handle"` notation (`quit_displays`)
   2. We refer to `"result"` in function names, but `"id"` in arguments. They're the same.
2. Add more type conversion routines. Also rationalise the existing routines. We should have a completely orthoganal set of interrogation function so that manipulation contigs, types and ids are the same.
3. Document usage of registration scheme within the contig comparitor (it's not straight forward or immediately obvious).

7 Writing Packages

An important feature of the newer Tcl/Tk based applications is the ability to write extensions to directly add new functionality. These are typical add new commands onto the main menus.

The best method of explaining the process of creating an extension is to work through an example. Here we supply the full code for a Gap4 extension to count base composition. The same techniques will apply to writing extensions for other programs and we will point out the Gap4 specific components. The example is somewhat simplistic, but hopefully will explain the framework needed to write a more complex package.

The complete sources for the composition package can be found in the '`src/composition`' file.

7.1 Creating a New Tcl Command

In general, for speed we wish our main algorithm to be written in C. Tcl is an interpreted language and runs very much slower than compiled C. As Tcl provides a method to extend the language with our own commands we will create a new command, which in this case is to be named "composition".

7.1.1 Registering the Command

Firstly we need to tell the Tcl interpreter which Tcl command should call which C function. We do this using the `Tcl_CreateCommand` function. This is typically called within the package initialisation routine. For a package named `composition` this is the `Composition_Init` routine.

```
/*
 * This is called when the library is dynamically linked in with the calling
 * program. Use it to initialise any tables and to register the necessary
 * commands.
 */
int Composition_Init(Tcl_Interp *interp) {
    if (NULL == Tcl_CreateCommand(interp,
                                  "composition",
                                  tcl_composition,
                                  (ClientData) NULL,
                                  (Tcl_CmdDeleteProc *) NULL)) {
        return TCL_ERROR;
    }

    return TCL_OK;
}
```

In the above example we are saying that the Tcl command '`composition`' should call the C function '`tcl_composition`'. If we wished to call the C function with a specific argument that is known at the time of this initialisation then we would specify it in the `ClientData` argument (`NULL` in this example). The full information on using `Tcl_CreateCommand` is available in the Tcl manual pages.

7.1.2 Parsing the Arguments

Our policy is to have a simple function to parse the command line arguments passed from Tcl. This should massage the arguments into a format usable by a separate (from Tcl) C or Fortran function which does the actual work. This clearly separates out the Tcl interface from the algorithms. The parsing will be done in the function registered with the Tcl interpreter. In our example this is `tcl_composition`.

The latest Tcl/Tk release provides functions for easing the parsing of command line arguments. In the future we *may* switch to using this scheme, but at present we use (and document) our own methods. A quick overview of this is that we declare a structure to hold the argument results, a structure to define the available command line parameters, and then call the `parse_args` or `gap_parse_args` function. Note that it is entirely up to the author of the package code for the arguments should be processed.

Firstly we need to include the '`cli_arg.h`' file. Secondly declare a structure containing the argument results. The structure does not need to be referenced outside of this file and so need not be in a public header file. Next we need a structure of type `cli_args[]` to specify the mapping of command line argument strings to argument result addresses. The `cli_args` structure is defined as follows.

```
        typedef struct {
        char *command;      /* What to recognise, including the '-' symbol */
        int type;           /* ARG_??? */
        int value;          /* Set if this argument takes an argument */
        char *def;          /* NULL if non optional argument */
        int offset;         /* Offset into the 'result' structure */
    } cli_args;
```

Command is a text string holding the option name, such as "-file". The last entry in the argument array should have a *command* of `NULL`.

Value is either 0 or 1 to indicate whether an extra argument is required after the command line option. A value of 1 indicates that an extra argument is needed.

Type specifies the type of this extra argument. It can be one of `ARG_INT`, `ARG_STR`, `ARG_ARR`, `ARG_FLOAT` and (for Gap4 only) `ARG_IO` to represent types of `int`, `char *`, `char []`, `float` and `GapIO *`. An option with no extra argument must have the type of `ARG_INT` as in this case the stored value will be 0 or 1 to indicate whether the option was specified.

Of the above types, `ARG_ARR` requires a better description. Options of this type are character arrays where the option argument is copied into the array. The *value* field for this type only specifies the length of the array. Finally the `offsetofa` macro instead of the `offsetof` macro (see below) must be used for the *offset* structure field. This type will possibly be removed in the future in favour of keeping `ARG_STR`. For `ARG_STR` the result is a character pointer which is set to the option argument. This requires no bounds checking and can use the standard `offsetof` macro.

Def specifies the default value for this option. If the option takes no extra argument or if it takes an extra argument and no default is suitable, then `NULL` should be used. Otherwise `def` is a text string, even in the case of `ARG_INT` in which case it will be converted to integer if needed.

Offset specifies the location within the results structure to store the result. The `offsetof` macro can be used to find this location. An exception to this is the `ARG_ARR` type where the `offsetofa` macro needs to be used instead (with the same syntax).

For our composition package we will have the following two structures.

```
    typedef struct {
        GapIO *io;
        char *ident;
    } test_args;
    test_args args;
    cli_args a[] = {
        {"-io",       ARG_IO,  1, NULL, offsetof(test_args, io)},
        {"-contigs",  ARG_STR, 1, NULL, offsetof(test_args, ident)},
        {NULL,        0,       0, NULL, 0}
    };
```

So we have two command line options, -io and -contigs, both of which take extra arguments. These are stored in `args.io` and `args.ident` respectively. The last line indicates the end of the argument list.

Once we've defined the structures we can actually process the process the arguments This is done using either `parse_args` or `gap_parse_args`. The latter of these two is for Gap4 only and is the only one which understands the `ARG_IO` type. The functions take four arguments which are the address of the `cli_args[]` array, the address of the result structure, and the `argc` and `argv` variables. The functions returns -1 for an error and 0 for success.

```
        if (-1 == gap_parse_args(a, &args, argc, argv)) {
            return TCL_ERROR;
        }
```

7.1.3 Returning a Result

To return a result to Tcl the *interp->result* variable needs to be set. This can be done in a variety of ways including setting the result manually or using a function such as `Tcl_SetResult`, `Tcl_AppendResult`, `Tcl_ResetResult` or `Tcl_DStringResult`.

However the choice of which to use is not as obvious as may first appear. A cautionary tale will illustrate some of the easy pitfalls. The following points are not made sufficiently clear in John Ousterhouts Tcl and Tk book. Additionally the problems are real and have been observed in the development of Gap4.

Consider the case where we have many commands registered with the interpreter. One such example could be:

```
    int example(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
    {
        /* ... */

        sprintf(interp->result, "%d", some_c_func());
        return TCL_OK;
    }
```

Now deep within `some_c_func` we have a `Tcl_Eval` call which happens to end with something like the following:

```
    proc some_tcl_func {} {
        # ...

        set fred jim
    }
```

Due to the call of `Tcl_Eval` in `some_c_func` the *interp->result* is now set to the last returned result, which is from the `set` command. In the above example *interp->result* points to 'jim'. The `sprintf` command in the `example` function will overwrite this string and hence change the value of the *fred* Tcl variable. This causes confusion and in some cases may also cause memory corruption where data is incorrectly freed.

The moral of this tale is to be extremely wary. As there is no knowledge of what `some_c_func` does (and remember it may get updated later) we seem to trapped. One possible solution is to rewrite the `example` function as follows.

```
int example(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
{
    int ret;
    /* ... */

    ret = some_c_func();
    Tcl_ResetResult(interp);
    sprintf(interp->result, "%d", ret);
    return TCL_OK;
}
```

This leads to another pitfall. If we have '`sprintf(interp->result, "%d", some_c_func(interp));`' and `some_c_func` calls (possibly indirectly) the `Tcl_ResetResult` function then we'll be modifying the *interp->result* address. This leads to undefined execution of code. (Is `sprintf` passed the original or final *interp->result* pointer?)

Therefore I'm inclined to think that we should never use `Tcl_ResetResult` except immediately before a modification of *interp->result* in a separate C statement. My personal recommendation is to never write directly to *interp->result*. Additionally never reset *interp->result* to a new string unless *interp->freeProc* is also updated correctly. In preference, use `Tcl_SetResult`.

The `Tcl_SetResult` function should always work fine, however it does not take `printf` style arguments. We have implemented a `vTcl_SetResult` which takes an *interp* argument and the standard `printf` format and additional arguments. For instance we would rewrite the example function as the following

```
int example(ClientData clientData, Tcl_Interp *interp, int argc, char **argv)
{
    int ret;
    /* ... */

    vTcl_SetResult(interp, "%d", some_c_func());
    return TCL_OK;
}
```

As a final note on `vTcl_SetResult`; the current implementation only allows strings up to 8192 bytes. This should be easy to remedy if it causes problems for other developers.

7.1.4 Writing the Code Itself

The final C code itself is obviously completely different for each extension.

In the example composition package we loop through each contig listed in our `-contigs` command line argument running a separate function that returns a Tcl list containing the total number of characters processed and the number of A, C, G, T and unknown nucleotides. Each list in turn is then added as an item to another list which is used for the final result.

```
    /* Do the actual work */
    Tcl_DStringInit(&dstr);
    for (i = 0; i < num_contigs; i++) {
        result = doit(args.io, contigs[i].contig, contigs[i].start,
                      contigs[i].end);
        if (NULL == result) {
            xfree(contigs);
            return TCL_ERROR;
        }

        Tcl_DStringAppendElement(&dstr, result);
    }

    Tcl_DStringResult(interp, &dstr);

    xfree(contigs);
    return TCL_OK;
}
```

The above is the end of the `tcl_composition` function. `doit` is our main algorithm written in C (which has no knowledge of Tcl). We use the Tcl dynamic strings routines to build up the final return value. The complete C code for this package can be found in the appendices.

If a command has persistent data about a contig (such as a plot containing the composition) the registration scheme should be used to keep this data up to date whenever database edits are made. See Chapter 6 [Gap4 Contig Registration Scheme], page 99.

7.2 Adding a GUI to the Command

Now we've defined a new Tcl command to perform the real guts of our example package we need to add Tk dialogues to provide a graphical interface to the user. This will typically be split into two main parts; the construction of the dialogues and the 'OK' callback procedure.

7.2.1 The Dialogue Creation

Firstly, we need to create the dialogue. This is done using both standard Tk commands and extra widgets defined in the tk_utils package. For the composition package the dialogue procedure is as follows.

```
    proc Composition {io} {
        global composition_defs

        # Create a dialogue window
        set t [keylget composition_defs COMPOSITION.WIN]
        if [winfo exists $t] {
            raise $t
            return
        }
        toplevel $t

        # Add the standard contig selector dialogues
        contig_id $t.id -io $io
        lorf_in $t.infile [keylget composition_defs COMPOSITION.INFILE] \
            "{contig_id_configure $t.id -state disabled}
             {contig_id_configure $t.id -state disabled}
             {contig_id_configure $t.id -state disabled}
             {contig_id_configure $t.id -state normal}
            " -bd 2 -relief groove
        # Add the ok/cancel/help buttons
        okcancelhelp $t.but \
            -ok_command "Composition2 $io $t $t.id $t.infile" \
            -cancel_command "destroy $t" \
            -help_command "show_help %composition Composition"

        pack $t.infile $t.id $t.but -side top -fill both
    }
```

Firstly we define the procedure name. In this case we'll call it `Composition`. It takes a single argument which is the IO handle of an opened Gap4 database.

Next we need to create a new window. We've stored the Tk pathname of this window in the `COMPOSITION.WIN` keyed list value in the defaults for this package. As our package is called *composition* the defaults are *composition_defs*. We define them as global and use `keylget` to fetch the window pathname. It is wise to check that the dialogue window doesn't already exist before attempting to create a new one. This could happen if the user selects the option from the main menu twice without closing down the first dialogue window.

Then the real dialogue components are added. In this case these consist of `contig_id`, `lorf_in` and `okcancelhelp` widgets. These are explained (FIXME: will be...) in the tk_utils and gap4 chapters. Note that the `okcancelhelp` command requires three Tcl scripts to execute when each of the Ok, Cancel and Help buttons are pressed.

For the Ok button we call the `Composition2` procedure with the widget pathnames containing the users selections. The Cancel button is easy as we simply need to destroy the dialogue window. The Help button will call the `show_help` command to display the appropriate documentation. More on this later.

7.2.2 Calling the New Command

Once the Ok callback from the `okcancelhelp` widget in the main dialogue has been executed we need to process any options the user has changed within the dialogue and pass these on to the main algorithms.

For the extension widget we set the OK callback to execute a `Composition2` procedure. This starts as follows.

```
    # The actual gubbins. This can be either in straight tcl, or using Tcl and
    # C. In this example, for efficiency, we'll do most of the work in C.
    proc Composition2 {io t id infile} {
        # Process the dialogue results:
        if {[lorf_in_get $infile] == 4} {
            # Single contig
            set name [contig_id_gel $id]
            set lreg [contig_id_lreg $id]
            set rreg [contig_id_rreg $id]
            SetContigGlobals $io $name $lreg $rreg
            set list "{$name $lreg $rreg}"
        } elseif {[lorf_in_get $infile] == 3} {
            # All contigs
            set list [CreateAllContigList $io]
        } else {
            # List or File of contigs
            set list [lorf_get_list $infile]
        }

        # Remove the dialogue
        destroy $t

        # Do it!
        SetBusy
        set res [composition -io $io -contigs $list]
        ClearBusy
```

For this Gap4 command we have used the `lorf_in` widget to let the user select operations for a single contig, all contigs, a list of contigs, or a file of contigs. We firstly process this to build up the appropriate values to send to the `-list` option of the `composition` Tcl command. The processes involved here are explained in the `lorf_in` widget documentation. (FIXME: to write).

Next we remove the dialogue window, enable the busy mode to grey out other menu items, and execute the command itself saving its result in the Tcl *res* variable.

The procedure then continues by stepping through the *res* variable using tcl list and formatting commands to output to the main text window with the `vmessage` command. The complete code for this can be found in the appendices.

7.2.3 The tclIndex file

One final requirement before the Tcl dialogue is complete is to create the '`tclIndex`' file.

Tcl uses a method whereby Tcl files are only loaded and executed when a command is first needed. This is done by referencing *auto_index* array in the Tcl error handler. This handler requires the '`tclIndex`' files to determine the location of each command. Failing to create this file will cause Tcl to complain that a command does not exist.

To create a '`tclIndex`' file start up either `stash` or `tclsh` and type '`auto_mkindex` *dir*' where *dir* is the name of the directory (often simply `"."`) containing the Tcl files. For the composition package this created the following '`tclIndex`' file.

```
# Tcl autoload index file, version 2.0
# This file is generated by the "auto_mkindex" command
# and sourced to set up indexing information for one or
# more commands.  Typically each line is a command that
# sets an element in the auto_index array, where the
# element name is the name of a command and the value is
# a script that loads the command.

set auto_index(Composition) [list source [file join $dir composition.tcl]]
set auto_index(Composition2) [list source [file join $dir composition.tcl]]
```

7.3 Creating the Config File

The package needs to have a config file - an '`rc`' file. For the composition package this will be named '`compositionrc`'. The file contains package dependencies, menu commands, and any user adjustable defaults.

For the composition package we do not need any dependencies. The package depends on gap4 and tk_utils, but both of these are already loaded. If we did need to use an additional package, or simply an additional dynamic library, then we could add further `load_package` commands to the start of the file.

Next we define the menu items. We could add an entirely new menu if the package defines many additional commands. In this example we'll simply add an extra command onto the standard Gap4 View menu.

```
# We want to add to the View menu a new command named "Test Command".
# This will call our TestCommand procedure with the contents of the global
# $io variable (used for accessing the gap database).
#
# The command itself should be greyed out when the database is not open or
# is empty.

add_command     {View.List Composition}       8 10 {Composition \$io}
```

This specifies that the `Composition $io` command is to be added to the View menu as 'List Composition'. It will be enabled only when the database is open and has data (8) and is disabled during busy modes and when the database has no data or is not open (10).

Next we add any defaults. For the composition package this is simply the dialogue values for the composition command.

```
# Now for the default values required by the composition command. Some of
# these are the sort of things that will be configured by users (eg the
# default cutoff score in a search routine) by creating their own .rc file
# (.compositionrc in this case). Others are values used entirely by the
# package itself. In our case  that's all we've got.

set_defx defs_c_in      WHICH.NAME      "Input contigs from"
set_defx defs_c_in      WHICH.BUTTONS   {list file {{all contigs}} single}
set_defx defs_c_in      WHICH.VALUE     3
set_defx defs_c_in      NAME.NAME       "List or file name"
set_defx defs_c_in      NAME.BROWSE     "browse"
set_defx defs_c_in      NAME.VALUE      ""

set_def COMPOSITION.WIN .composition
set_def COMPOSITION.INFILE $defs_c_in
```

7.4 Writing the Online Help

The online help (including this) and printed manual for our programs are written using Texinfo. However due to the usage of pictures (which aren't supported by Texinfo) we've made several modifications to the documentation system. We have modified makeinfo and texi2html scripts too. Consequently the system we use for documentation is not ready for public usage.

However the final files needed for online usage by the applications can be produce by any system capable of creating HTML files and our own '`.index`' and '`.topic`' files.

The principle method of bringing up help from a package is to use the `show_help` command. For the composition widget we used the following.

```
show_help %composition Composition
```

The `%composition` indicates that the `show_help` command should read the '`composition.topic`' and '`composition.index`' files. These are normally read from the '`$STADENROOT/manual/`' directory, but by preceeding the name with a percent sign we can direct the `show_help` command to search for these files in the composition package directory.

The last argument of `show_help` is the topic to display. In this case it is `Composition`. If the topic includes spaces then remember to use the Tcl quoting mechanism. The topic file is then scanned to find and line with this topic as the first 'word'. The second 'word' contains the index name. The index name is then looked up in the index file (as the first word) to find the URL (the second word). This two stage lookup is designed to protect against renaming section headings in the documentation. The index file can be easily created by parsing the html files to generate a mapping of heading names to URLs. However if the documentation changes we do not wish to need to change the Tcl calls to `show_help`.

```
composition.topic file:

    {Composition} {Composition}



composition.index file:

    {Composition} composition.html
```

For the composition package we have very simple topic and index files. The index and topic names are identical, so the topic file is trivial. The index file contains a single line mapping the `Composition` index entry to the `composition.html` file. If a named tag within the html file is needed then the URL would be `composition.html#tagname`. The html file itself is held within the same directory as the topic and index files.

7.5 Wrapping it all up

We've now got all the code that we need to build a complete package. If this package is to be kept separate from the main Staden Package installation tree then we need to build our own directory tree for the package.

For example, we'll create a separate directory for the composition package named '`/home/spackages`'. Within this directory we should place the rc file ('`compositionrc`'), the documentation ('`composition.topic`', '`composition.index`' and '`composition.html`') and the Tcl files '`composition.tcl`' and '`tclIndex`'.

Additionally we need to have a dynamic library containing the C command. This should be placed in '`/home/spackages/MACHINE-binaries/`' where '`MACHINE`' is the machine type (eg `alpha`, `solaris`, `sun`, `sgi`, `linux` or `windows`). The library will probably be named something like '`libcomposition.so`'.

The actual compilation of the library is complicated due to each machine type having different linker options. The full description of the Makefile system is beyond the scope of this documentation, but in brief, the system works by having a single '`Makefile`' for the package, a '`global.mk`' file in the '`$STADENROOT/src/mk`' directory containing general definitions, and a system specific (eg '`alpha.mk`') file also in '`$STADENROOT/src/mk`' defining system architecture specific definitions. These combine to allow system independent macros to be used for building dynamic libraries. The complete composition package Makefile is in the appendices.

Once the package has been installed correctly an `ls -R` on the installation directory should look something like the following.

```
alpha-binaries/     composition.index   composition.topic   tclIndex
composition.html    composition.tcl     compositionrc

./alpha-binaries:
libcomposition.so   so_locations
```

Note that packages for multiple architectures may share the same installation tree as each architecture will need only its own '`MACHINE-binaries`' directory.

The final requirement is to add the package onto gap4. This is done by adding the following to the users '`.gaprc`' file (where '`/installation/directory/`' is the location where containing the list of files).

```
load_package /installation/directory/composition
```

Appendix A Composition Package

Here are the main source components for the Gap4 composition extension

A.1 Makefile

```
# Makefile for the composition 'package' to add to gap4.

LIBS = composition
PROGS= $(LIBS)

SRCROOT=$(STADENROOT)/src
include $(SRCROOT)/mk/global.mk
include $(SRCROOT)/mk/$(MACHINE).mk

INSTALLDIR  = ./install

INCLUDES_E += $(TCL_INC) $(TKUTILS_INC) $(GAP4_INC) $(G_INC)
CFLAGS     += $(SHLIB_CFLAGS)

TESTBIN     = $(O)
L           = $(INSTALLDIR)/$(O)

# Objects
OBJS = \
        $(TESTBIN)/composition.o

DEPS = \
        $(G_DEP) \
        $(TKUTILS_DEP) \
        $(TCL_DEP)

#
# Main dependency
#
$(LIBS) : $(L)/$(SHLIB_PREFIX)$(LIBS)$(SHLIB_SUFFIX)
        @

$(L)/$(SHLIB_PREFIX)$(LIBS)$(SHLIB_SUFFIX): $(OBJS)
        -mkdir $(INSTALLDIR)
        -mkdir $(INSTALLDIR)/$(O)
        $(SHLIB_LD) $(SHLIB_LDFLAGS) $@ $(OBJS) $(DEPS)

DEPEND_OBJ = $(OBJS)

install: $(LIBS)
        cp tclIndex composition.tcl compositionrc composition.topic \
        composition.index composition.html $(INSTALLDIR)

include dependencies
```

A.2 composition.c

```
#include <tcl.h>

#include "IO.h"                 /* GapIO */
#include "gap_globals.h"        /* consensus/quality cutoffs */
#include "qual.h"               /* calc_consensus() */
#include "cli_arg.h"            /* cli_arg, parse_args() */

static int tcl_composition(ClientData clientData, Tcl_Interp *interp,
                           int argc, char **argv);
static char *doit(GapIO *io, int contig, int lreg, int rreg);

/*
 * This is called when the library is dynamically linked in with the calling
 * program. Use it to initialise any tables and to register the necessary
 * commands.
 */
int Composition_Init(Tcl_Interp *interp) {
    if (NULL == Tcl_CreateCommand(interp,
                                  "composition",
                                  tcl_composition,
                                  (ClientData) NULL,
                                  (Tcl_CmdDeleteProc *) NULL))
        return TCL_ERROR;

    return TCL_OK;
}


/*
 * The composition itself.
 * This is called with an argc and argv in much the same way that main()
 * is. We can either parse them ourselves, our use the gap parse_args
 * utility routine.
 */
static int tcl_composition(ClientData clientData, Tcl_Interp *interp,
                           int argc, char **argv) {
    int num_contigs;
    contig_list_t *contigs = NULL;
    char *result;
    int i;
    Tcl_DString dstr;

    /* A structure definition to store the arguments in */
    typedef struct {
        GapIO *io;
        char *ident;
    } test_args;

    /* The mapping of the argument strings to our structure above */
    test_args args;
    cli_args a[] = {
        {"-io",       ARG_IO,  1, NULL, offsetof(test_args, io)},
        {"-contigs",  ARG_STR, 1, NULL, offsetof(test_args, ident)},
        {NULL,      0,       0, NULL, 0}
    };

    /*
     * First things first, add a header to the output window. This shows the
     * date and function name.
     */
    vfuncheader("test command");

    /* Parse the arguments */
    if (-1 == gap_parse_args(a, &args, argc, argv)) {
        return TCL_ERROR;
    }

    active_list_contigs(args.io, args.ident, &num_contigs, &contigs);
    if (num_contigs == 0) {
        xfree(contigs);
        return TCL_OK;
    }

    /* Do the actual work */
    Tcl_DStringInit(&dstr);
    for (i = 0; i < num_contigs; i++) {
        result = doit(args.io, contigs[i].contig, contigs[i].start,
                      contigs[i].end);
        if (NULL == result) {
            xfree(contigs);
            return TCL_ERROR;
        }

        Tcl_DStringAppendElement(&dstr, result);
    }

    Tcl_DStringResult(interp, &dstr);

    xfree(contigs);
    return TCL_OK;
}

/*
 * Our main work horse. For something to do as an example we'll output
 * the sequence composition of the contig in the given range.
 */
static char *doit(GapIO *io, int contig, int lreg, int rreg) {
    static char result[1024];
    char *consensus;
    int i, n[5];

    if (0 == lreg && 0 == rreg) {
        rreg = io_clength(io, contig);
        lreg = 1;
    }

    if (NULL == (consensus = (char *)xmalloc(rreg-lreg+1)))
        return NULL;

    if (-1 == calc_consensus(contig, lreg, rreg, CON_SUM,
                             consensus, NULL, NULL, NULL,
                             consensus_cutoff, quality_cutoff,
                             database_info, (void *)io)) {
        xfree(consensus);
        return NULL;
    }

    n[0] = n[1] = n[2] = n[3] = n[4] = 0;
    for (i = 0; i <= rreg - lreg; i++) {
        switch(consensus[i]) {
        case 'a':
        case 'A':
            n[0]++;
            break;

        case 'c':
        case 'C':
            n[1]++;
            break;

        case 'g':
        case 'G':
            n[2]++;
            break;

        case 't':
        case 'T':
            n[3]++;
            break;

        default:
            n[4]++;
        }
    }

    /* Return the information */
    sprintf(result, "%d %d %d %d %d %d",
            rreg - lreg + 1, n[0], n[1], n[2], n[3], n[4]);

    xfree(consensus);

    return result;
}
```

A.3 composition.tcl

```
# The main command procedure to bring up the dialogue
proc Composition {io} {
    global composition_defs

    # Create a dialogue window
    set t [keylget composition_defs COMPOSITION.WIN]
    if [winfo exists $t] {
        raise $t
        return
    }
    toplevel $t

    # Add the standard contig selector dialogues
    contig_id $t.id -io $io
    lorf_in $t.infile [keylget composition_defs COMPOSITION.INFILE] \
        "{contig_id_configure $t.id -state disabled}
         {contig_id_configure $t.id -state disabled}
         {contig_id_configure $t.id -state disabled}
         {contig_id_configure $t.id -state normal}
        " -bd 2 -relief groove

    # Add the ok/cancel/help buttons
    okcancelhelp $t.but \
        -ok_command "Composition2 $io $t $t.id $t.infile" \
        -cancel_command "destroy $t" \
        -help_command "show_help %composition Composition"

    pack $t.infile $t.id $t.but -side top -fill both
}

# The actual gubbins. This can be either in straight Tcl, or using Tcl and
# C. In this example, for efficiency, we'll do most of the work in C.
proc Composition2 {io t id infile} {
    # Process the dialogue results:
    if {[lorf_in_get $infile] == 4} {
        # Single contig
        set name [contig_id_gel $id]
        set lreg [contig_id_lreg $id]
        set rreg [contig_id_rreg $id]
        SetContigGlobals $io $name $lreg $rreg
        set list "{$name $lreg $rreg}"
    } elseif {[lorf_in_get $infile] == 3} {
        # All contigs
        set list [CreateAllContigList $io]
    } else {
        # List or File of contigs
        set list [lorf_get_list $infile]
    }

    # Remove the dialogue
    destroy $t

    # Do it!
    SetBusy
    set res [composition -io $io -contigs $list]
    ClearBusy

    # Format the output
    set count 0
    set tX 0
    set tA 0
    set tC 0
    set tG 0
    set tT 0
    set tN 0
    foreach i $res {
        vmessage "Contig [lindex [lindex $list $count] 0]"
        incr count

        set X [lindex $i 0]; incr tX $X
        if {$X <= 0} continue;

        set A [lindex $i 1]; incr tA $A
        set C [lindex $i 2]; incr tC $C
        set G [lindex $i 3]; incr tG $G
        set T [lindex $i 4]; incr tT $T
        set N [lindex $i 5]; incr tN $N
        vmessage "  Length  [format %6d $X]"
        vmessage "  No. As  [format {%6d %5.2f%%} $A [expr 100*${A}./$X]]"
        vmessage "  No. Cs  [format {%6d %5.2f%%} $C [expr 100*${C}./$X]]"
        vmessage "  No. Gs  [format {%6d %5.2f%%} $G [expr 100*${G}./$X]]"
        vmessage "  No. Ts  [format {%6d %5.2f%%} $T [expr 100*${T}./$X]]"
        vmessage "  No. Ns  [format {%6d %5.2f%%} $N [expr 100*${N}./$X]]\n"
    }

    if {$count > 1} {
        vmessage "Total length [format %6d $tX]"
        vmessage "Total As     [format {%6d %5.2f%%} $tA [expr 100*${A}./$tX]]"
        vmessage "Total Cs     [format {%6d %5.2f%%} $tC [expr 100*${C}./$tX]]"
        vmessage "Total Gs     [format {%6d %5.2f%%} $tG [expr 100*${G}./$tX]]"
        vmessage "Total Ts     [format {%6d %5.2f%%} $tT [expr 100*${T}./$tX]]"
        vmessage "Total Ns     [format {%6d %5.2f%%} $tN [expr 100*${N}./$tX]]"
    }
}
```

Function Index

This index contains lists of the C and Tcl function calls available. Entry items listed with a *(T)* suffix are callable from Tcl. Entry items listed with a *(C)* suffix are callable from within C.

Variable and Type Index

This index contains lists of C and Tcl variables and types. Entry items listed with a *(T)* suffix are Tcl variables. Entry items listed with a *(C)* suffix are C variables. All types are C.

Concept Index

Short Contents

Table of Contents