SUN/186.3

Starlink Project
Starlink User Note 186.3

D. L. Terrett

3 March 1999

STARTCL — Starlink Extensions to Tcl & Tk version 1.3 User's Guide

---

Abstract

The STARTCL package is a set of extensions to the Tcl/Tk language to enable Tcl and Tk applications to display Starlink graphics inside widgets embedded in the application, send and receive ADAM messages and access ADAM noticeboards.

This document will be of interest to programmers writing graphics user interfaces for Starlink applications in Tcl/Tk.

Contents

1 Introduction

A powerful public domain utility called Tcl/Tk ("Tcl and the Tk Toolkit", John K. Ousterhout, Addison-Wesley, 1994) is coming into wide use as a command language and user interface builder.

This note describes three extensions to Tcl and the Tk Toolkit[1] that enable Tcl/Tk applications to cooperate with Starlink applications. The extensions are:

- A Tk widget that displays Starlink graphics by emulating the "gwm" graphics window server and a "gwm" canvas item.
- An interface to the Adam message system.
- An interface to the Adam notice board system.

  The message system and notice board extensions are both based on work done by Dennis Kelly of the Royal Observatory Edinburgh.

A version of the Tcl shell (called `atclsh`) and the windowing shell (called `awish`) with all of these extension built in are provided which are useful for trying them out. However, for production systems, a bespoke version of the shell with a subset of the available extensions (both from Starlink and elsewhere) will normally be required.

The Starlink extensions can be built into a Tcl or Tk application by including one or more of the following functions in `tclAppInit.c` or `tkAppInit.c`.

Extension	Function	include file	object module
Adam	`Tcladam_Init(interp)`	`tclAdam.h`	`tclAdam.o`
Gwm	`Tkgwm_Init(interp)`	`tkGwm.h`	`tkGwm.o`
Nbs	`Tclnbs_Init(interp)`	`tclNbs.h`	`tclNbs.o`

where `interp` is a pointer to a `Tcl_Interp`. All three functions return a Tcl status value. The application should then be linked with the specified object module all of which can be found in `/star/lib`.

Alternatively, the Adam and Gwm extensions can be loaded at run time into the standard interpreters with `load` commands of the form:

```
load /star/lib/libtclAdam.so tclAdam
load /star/lib/libtkGwm.so tkGwm
```

The disadvantages of this method is the need to supply a path name for the shareable library which makes the script dependent on the location of the starlink software and that this mechanism may not be available on all operating systems in the future.

[1] These extensions will only work with Tk Version 4.2 or later

2 The GWM Extension

A Graphics Window Manager (GWM) widget is a window that conventional (*i.e.* non-event driven) programs that use Starlink graphics packages (*e.g.*, GKS, SGS, PGPLOT, NCAR, IDI *etc.*) can use to plot in on X windows displays. GWM is described in SUN/130. The gwm widget makes it possible to integrate GKS and IDI based graphics with user interfaces written in Tcl and Tk. It also has commands for dumping its contents in JPEG, PostScript or LJ250 inkjet format, changing the colour lookup table, clearing the window, scrolling the window and displaying a cross hair that covers full width and height of the window.

The GWM canvas item turns an area of a canvas widget into a GWM window; this provides similar functionality to the GWM widget but also allows other canvas items (lines, polygons, *etc.*) to be overlayed on the window and therefore graphics from external programs and element created and minipulated by Tk to be mixed.

2.1 The GWM Widget

2.1.1 WIDGET COMMAND

The **gwm** command creates a new Tcl command whose name is *pathName*. The command may then be used to perform various operations on the widget. It has the following general form:

> **gwm** *pathName option ?arg arg...?*

Option and *args* determine the exact behaviour of the command.

The following widget commands are possible for gwm widgets:

pathName ***cget*** *option*

> Returns the current value of the configuration option given by *option*. *option* may have any of the values accepted by the gwm command.

pathName ***clear***

> Clear the picture in the widget by setting the entire window to pixel value zero.

pathName ***configure*** *?option? ?value option value...?*

> Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName* (see **Tk_ConfigureInfo** for information on the format of this list). If *option* is specified with no *value*, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no *option* is specified). If one or more *option-value* pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. *Option* may have any of the values accepted by the **gwm** command.

pathName ***get colour*** *index*

Returns the colour of colour table entry *index* as a text string in format *#RRRRGGGGBBBBB*. *Index* -1 returns the colour of the overlay.

pathName ***ovclear***

Clear the overlay.

pathName ***print*** *filename*

Write the contents of the widget to *filename* in the format specified by the *printformat* option. This command returns as soon as the printing has started; the generation of the file continues whenever the Tcl interpreter is idle. When the printing is complete, the global variable specified by the *printvar* option is set to 1.

pathName ***set colour*** *index colour*

Set the colour of colour table index *index* to em colour. *Index* -1 sets the colour of the overlay plane.

pathName ***set crosshair*** *x_position y_position*

Sets the XY position of the crosshair. The values may be in any of the forms acceptable to **Tk_GetPixels**.

2.1.2 Printing

The gwm widget has a command for printing the contents of the gwm window in a number of formats (JPEG, various flavours of PostScript and HP inkjet). The process of generating the file can be quite time consuming for a large window so the print command returns control as soon as the window contents have been captured and the output file created. Actual generation of the file is then carried out whenever the Tcl interpreter is idle enabling the application to continue to respond to X events while the printing is in progress. Completion of the printing process is signaled by a global Tcl variable (**gwm_printvar** by default) being set to 1.

The simplest way to detect completion is to wait for the value of the variable to change. *e.g.*:

```
global gwm_printvar
set gwm_printvar 0
if [catch {.gwm print plot.ps}] {
   ...an error
} else {
   tkwait variable gwm_printvar
}
```

An alternative approach is to set a trace on the variable so that a procedure is invoked when the print completes. *e.g.*:

```
proc done {name1 name2 opt} {
    trace vdelete gwm_printvar w done
}

global gwm_printvar
set gwm_printvar 0
if [catch {.gwm print plot.ps}] {
    ...an error
} else {
    trace variable gwm_printvar w done
}
```

In this example the trace routine merely deletes itself but any other processing can be done here.

2.1.3 WIDGET OPTIONS

Name: **background**
Class: **Background**
Command-Line Switch: **-background** or **-bg**

Specifies the background colour (colour table entry 0) of the gwm window. Defaults to Black.

Name: **colours**
Class: **Colours**
Command-Line Switch: **-colours**

Specifies the number of colours that the gwm window will attempt to allocate for its colour table. Setting this option after the widget has been created has no effect. Defaults to 64.

Name: **crosshair**
Class: **Crosshair**
Command-Line Switch: **-crosshair**

Specifies whether the cross-hair should be displayed.

Name: **crossColour**
Class: **CrossColour**
Command-Line Switch: **-crosscolour**

Specifies the colour of the crosshair.

Name: **cursor**
Class: **Cursor**
Command-Line Switch: **-cursor**

Specifies the mouse cursor to be used for the widget. The value may have any of the forms acceptable to **Tk_GetCursor**.

Name: **foreground**
Class: **Foreground**
Command-Line Switch: **-foreground** or **-fg**

Specifies the foreground colour (colour table entry 1) of the gwm window. Defaults to White.

Name: **name**
Class: **Name**
Command-Line Switch: **-name** or **-gwmname**

Specifies the name of the gwm window. Setting this option after the widget has been created has no effect. The gwm name is the name used by graphics programs to connect to the window. Defaults to "xwindows".

Name: **height**
Class: **Height**
Command-Line Switch: **-height**

Specifies the width of the gwm widget. The value is in any of the forms acceptable to **Tk_GetPixels**. Changing this option after the widget has been created has no effect. Defaults to 512 pixels.

Name: **jpegQuality**
Class: **JpegQuality**
Command-Line Switch: **-jpegquality**

Specifies the quality of the JPEG image in the range 0-100 where 0 is the lowest quality/highest compression and 100 is the highest quality/lowest compression. The normally useful range is 50-95, values above 95 give dramatically increased file size with practically no increase in quality while below 50 the image quality degrades the image quality without much additional compression.

Name: **jpegProgressive**
Class: **JpegProgressive**
Command-Line Switch: **-jpegprogressive**

Specifies a boolean value that determines whether the JPEG image is "progressive". Progressive images are useful for use on world wide web servers as it allows a browser to display a low quality image quickly and then improve the quality as more data arrives. However not all browsers and other image reading software can interpret progressive JPEGs.

Name: **minColours**
Class: **MinColours**
Command-Line Switch: **-mincolours**

Specifies the minimum number of colours that the gwm window should allocate for its colour table. If fewer than this number of colours are available, the creation of the widget will fail. Changing this option after the widget has been created has no effect. Defaults to 2.

Name: **overlay**
Class: **Overlay**
Command-Line Switch: **-overlay**

Specifies a boolean value that determines whether the gwm window will have an overlay plane. Changing this option after the widget has been created has no effect. Defaults to "no".

Name: **paperWidth**
Class: **PaperWidth**
Command-Line Switch: **-paperwidth**

Specifies the maximum width of the plot created by the **print** command. The value is in any of the forms acceptable to **Tk_GetPixels**. The plot will be the largest size possible that will fit on the paper (after rotation through 90 degrees if necessary) with the same aspect ratio as the gwm window. Defaults to 180mm (a little less than the width of an A4 page).

Name: **paperHeight**
Class: **PaperHeight**
Command-Line Switch: **-paperheight**

Specifies the maximum height of the plot in the same way as **paperWidth**. Defaults to 250mm (a little less than the height of an A4 page).

Name: **printBackground**
Class: **PrintBackground**
Command-Line Switch: **printbackground** or **-printbg**

Specifies the colour used for the background of a plot. Any pixel with value zero is plotted in the background colour. Defaults to White.

Name: **printFormat**
Class: **PrintFormat**
Command-Line Switch: **-printformat**

Specifies the format of the file produced by the **print** command. Valid values are:

ps postscript Monochrome PostScript.

colour_ps colour_postscript Colour PostScript.

eps bw_eps Monochrome encapsulated PostScript.

colour_eps Colour encapsulated PostScript.

HPinkjet LJ250 Inket.

JPEG JPEG

Name: **printVariable**
Class: **PrintVariable**
Command-Line Switch: **-printvariable**

Specifies the name of the global variable that is set when a print command has completed. The default is **gwm_printvar**.

Name: **takeFocus**
Class: **TakeFocus**
Command-Line Switch: **-takefocus**

See the options man page for a description of this option.

Name: **width**
Class: **Width**
Command-Line Switch: **-width**

Specifies the width of the gwm widget. The value is in any of the forms acceptable to **Tk_GetPixels**. Changing this option after the widget has been created has no effect.

Name: **xOffset**
Class: **XOffset**
Command-Line Switch: **-xoffset**

Specifies by how much the picture is offset from the left hand edge of the window. The value is in any of the forms acceptable to **Tk_GetPixels**. Positive values shift the picture to the right and negative values shift the picture to the right. The default is zero.

Name: **xOvOffset**
Class: **XOvOffset**
Command-Line Switch: **-xovoffset**

Specifies by how much the overlay is offset from the left hand edge of the window. The value is in any of the forms acceptable to **Tk_GetPixels**. Positive values shift the picture to the right and negative values shift the picture to the right. The default is zero.

Name: **yOffset**
Class: **YOffset**
Command-Line Switch: **-yoffset**

Specifies by how much the picture is offset from the top edge of the window. The value is in any of the forms acceptable to **Tk_GetPixels**. Positive values shift the picture to down the screen and negative values shift the picture to up the screen. The default is zero.

Name: **yOvOffset**
Class: **YOvOffset**
Command-Line Switch: **-yovoffset**

Specifies by how much the overlay is offset from the top edge of the window. The value is in any of the forms acceptable to **Tk_GetPixels**. Positive values shift the picture to down the screen and negative values shift the picture to up the screen. The default is zero.

2.1.4 Colourmaps

A gwm widget always uses the colourmap of its parent window; if there are insufficient colourmap entries left (less than **minColours**) the creation of the widget will fail. In these circumstances, a gwm widget can still be created by creating a frame or top level widget with a new colourmap (with the **-colormap new** option) and using this as the parent of the gwm widget[2].

In the case of a frame with a new colourmap it is also necessary to inform the window manager so that the correct colourmap is installed when the frame receives the input focus with:

```
wm colormapwindows [winfo toplevel $w] $w
```

where `w` has been set to the name of the new frame widget.

The same technique can be used to create a gwm widget with a visual type other than the default for the screen being used.

2.1.5 Tk Procedures

A number of Tk procedures can be found in `/star/lib/startcl` that may be useful when using gwm widgets either directly or as examples that can be modified to suit particular purposes. To use them directly, append `/star/lib/startcl` to the global variable **auto_path**, *e.g.*:

```
lappend auto_path /star/lib/startcl
```

These procedures make use of a global array called **gwm_priv**.

gwm_colourDialog *w gwm exit*

Creates a modeless dialog box with controls for modifying the colours of a gwm widget.

w A name to be used for the top level widget of the dialog box.

gwm The name of the gwm widget to be manipulated.

c The control used to pop up the dialog box. This control will be disabled when the dialog box is popped up and re-enabled when it is dismissed.

[2]Some existing graphics programs may attempt to write to the wrong colourmap when used with a gwm widget packed into a frame with a new colourmap and generate an X error. Relinking the program with the latest version of the Starlink software will correct this problem

gwm_gwmWithScroll *w args*

Returns a gwm widget with scroll bars for scrolling the window (and overlay plane if there is one).

w A name to be used for the frame widget that contains the gwm widget and the scroll bars.

args Additional arguments to the gwm widget creation command.

gwm_printDialog *w gwm*

Creates a modal dialog box containing controls for printing a gwm widget.

w A name to be used for the top level widget of the dialog box.

gwm The name of the gwm widget to be printed.

c The control used to pop up the dialog box. This control will be disabled when the print starts and re-enabled when it completes.

An gwm server that uses these procedures can be found in `/star/bin/startcl/gwm`. This is the default server for Starlink systems. It creates a server with buttons for changing the colours, printing the gwm window and clearing the picture and overlay planes. The coordinates of the mouse pointer are also displayed whenever it is over the gwm window. A listing of this script can be found in appendix A.

Arguments for the gwm widget creation command can be supplied on the command line (except **-name** which is interpreted as an **awish** option; use **-gwmname instead**).

To use your own gwm server by default whenever a gwm server is created (either with the xmake command or automatically by a graphics program), copy `/star/bin/startcl/gwmXrefresh` to a directory in your `PATH` that is positioned before `/star/bin/startcl` and edit line 13 so that the variable `gwmscript` is set to the location of suitable script that creates a gwm widget.

2.2 GWM Canvas item

A gwm canvas item is a rectangular area of a canvas widget that has been turned into a gwm window.

The canvas item duplicates much of the functionality of the gwm widget except that:

The canvas item does not support overlays.

Printing to JPEG or LJ250 format is not supported.

Printing is not done as a background activity so the Tk application freezes while the canvas is being printed.

Only one gwm item can exist on any particular canvas

There is no crosshair (although this can easily be implemented with line canvas items). widget.

The important additional functionality that the canvas item offers is the ability to overlay the gwm picture with other canvas item; since these items can be manipulated through Tcl procedures it is possible to write programs that enable the user to interact with graphics in a much wider variety of ways than is possible though GKS or IDI alone. However, any drawing done by an external program will overwrite any existing canvas items until the window is redrawn by Tk; the easiest way to force the window to be redrawn is to use the canvas `raise` or `lower` commands (for example, to "lower" the gwm canvas item so that it is underneath all other items).

Gwm items are created with widget commands of the following form:

pathName create gwm *x y option value option value ...*

The arguments *x* and *y* give the coordinates of the item's top left hand corner. After the coordinates there may be any number of *option-value* pairs, each of which sets one of the configuration options of the item. The same *option-value* pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for gwm items:

-background *colour*
-bg *colour*

Specifies the background colour (colour table entry 0) of the gwm window. Defaults to Black.

-colours *number*

Specifies the number of colours that the gwm window will attempt to allocate for its colour table. Setting this option after the item has been created has no effect. Defaults to 64.

-command *number*

Specifies the name of the command that is created to clear (erase the picture in) the item and modify the colour table. If no value is specified, no command will be created. Changing this option after the item has been created has no effect.

```
command clear
```

clears the item.

```
command set colour n colour
```

sets colour table entry *n* to *colour*.

```
command get colour n
```

returns the colour stored in colour table entry *n*.

-foreground *colour*
-fg *colour*

Specifies the foreground colour (colour table entry 1) of the gwm window. Defaults to White.

-gwmname *name*
-name *name*

Specifies the name of the gwm window. Setting this option after the item has been created has no effect. The gwm name is the name used by graphics programs to connect to the window. Defaults to "xwindows".

-height *height*

Specifies the width of the gwm item. The value is in any of the forms acceptable to **Tk_GetPixels**. Changing this option after the item has been created has no effect. Defaults to 512 pixels.

-mincolours *number*

Specifies the minimum number of colours that the gwm window should allocate for its colour table. If fewer than this number of colours are available, the creation of the item will fail. Changing this option after the item has been created has no effect. Defaults to 2.

-tags *taglist*

Specifies a set of tags to apply to the item. *Taglist* consists of a list of tag names, which replace any existing tags for the item. *Taglist* may be an empty list.

-width *width*

Specifies the width of the gwm item. The value is in any of the forms acceptable to **Tk_GetPixels**. Changing this option after the item has been created has no effect.

3 ADAM Message System Extension

The Adam message system extensions provide access to the message system at the level of the ams library and a fairly detailed knowledge of the Adam message system and how it is used by Adam tasks is required. Therefore, this section describes a set of Tcl procedures that use the message system interface to provide a higher level interface that is suitable for controlling Adam tasks from a graphical user interface. The low-level commands are described in appendix C.

Some of the features of Adam tasks described here appeared in release 3.1 and so you must ensure that any tasks to be used with Tcl have been linked with Adam 3.1 or later.

3.1 The Message System

Command languages (such as ICL or Tcl/Tk) control Adam tasks by sending them Adam messages and responding to any messages from the task The messages from the task request such things as new values for parameters or contain text to displayed to the user. This section describes what sort of messages tasks respond to and what replies can be expected; it concentrates on data reduction tasks and ignores some of the more complicated behavour of instrument control tasks.

3.1.1 The OBEY message

An adam task supports one or more commands, normally referred to as *actions*, (*e.g.* the KAPPA task supports "add", "cadd", "stats" etc.) and is instructed to execute one of them by sending it an "OBEY" message. The message contains the action name (*e.g.* "add") and a command line which can contain parameter values and any of the keywords that modify the behaviour of the parameter system (*e.g.* "in1=test accept"). If the task recognises the action name[3] it responds with an `actstart` message; the message name field of the message will be the action name and the status field will be `DTASK__ACTSTART`. The task then starts executing the command.

In the simplest possible case, where the action writes no text and does not need to prompt for any parameter values, the task completes the action and sends an `endmsg` message to the command language. The message name is again the action name and the status will be `DTASK__ACTCOMPLETE`. The task is now ready to execute another action.

If, as usually happens, the task does want to display some text, it will send one or more `inform` messages; the text to display will be in the *value* field of the message. Note that the message system makes no distinction between text and error reports, however, the adam error reporting system always prefixes the first line of an error message with two exclamation marks and subsequent lines with a single exclamation mark. If the task needs a value for a parameter it will send a `paramreq` message; the *value* field will contain the parameter name, prompt, default value etc.. The command language must reply with a `paramrep` message with the new parameter value in the *value* field.

The only other message that a task might send as a result of an "OBEY" is a `sync` message; the purpose of this message is to instruct the command language to ensure that all output has been actually written to the screen for the user to read—for example, instructions on how to perform some interaction with a graphics device. In the context of a GUI, where updates to the screen are performed asynchronously whenever the GUI process is idle, the message probably has no useful function. However, a `synchrep` message must be sent to acknowledge the `sync` message (the adamtask library described below does this automatically).

If an error occurs during the execution of a command, the *status* field of the final `endmsg` message will be the final exit status of the action; ie. something other than `DTASK__ACTCOMPLETE`. If the action cannot even be started (if, for example, the action name isn't recognised) there will be no `actstart` message and the `actcomplete` message will contain an error status (eg. `SUBPAR__NOACT`).

[3]Tasks that contain only one command ignore the action name and execute the command regardless

3.1.2 The SET message

To set a parameter, you send a "SET" message with the message *name* set to the parameter name in the form *action*:*parameter* (eg. stats:ndf) and the value field to the desired parameter value. The task will respond with a `setresponse` message with a status field set to `SAI__OK` if the parameter was sucessfully set. If not, it will send one of more `inform` messages containing error messages followed by a `setresponse` message with a status other than `SAI__OK` (for example, `SUBPAR__NOPAR` if the parameter didn't exist). When the task executes the coresponding action it will behave exactly as if the parameter had been set with a command line argument in the "OBEY" message.

3.1.3 The GET message

To get a parameter value, send a "GET" message with the parameter in the message `name` field; the task will respond with a `getresponse` message with the `name` set to the parameter name and with the parameter's value in the `value` field. Errors are handled in the same way as for a "SET". The "GET" message is only really useful if the task is running with its type set to "I", otherwise all of an actions parameters are annulled (set to have no value) when an action completes.

3.1.4 The CONTROL message

A "CONTROL" message instructs a task to perform one of the pre-defined actions that all Adam tasks support. There are two control actions recognised:

`default` instructs the task to change to a new default directory. The value field of the message should be set to the new directory. The task will respond with a `controlresponse` message with the value field set to the new directory. If a "CONTROL" message with the action set to "default" has a blank value field the value field in the `setresponse` message will be set to the current default directory.

`par_reset` sets all of the tasks parameters to the state they were in when the task was first loaded. The task responds with a `controlresponse` message.

3.1.5 The CANCEL message

The final type of message is the "CANCEL" message which is similar to an "OBEY" message but data reduction tasks do not support any "CANCEL" actions.

3.2 The Tcl/Tk Interface

The interface is loaded by reading the file `/star/lib/startcl/adamtask.tcl` with the tcl command:

```
source /star/lib/startcl/adamtask.tcl
```

This defines a procedure called `adamtask` which starts an Adam task. It takes two arguments: a name to be used to refer to the task in later Tcl commands, and, optionally, the name of the executable file to run. if the second argument is omitted, the name must be the name, as known

to the Adam message system, of a task already loaded. In addition it creates a new Tcl command with the same name as the task which can then be used to communicate with the task.

For example:

```
adamtask kappa /star/bin/kappa/kappa_mon
```

will start the KAPPA monolith and create a Tcl command called **kappa** which can be used to send messages to the task. The first argument to this new command is the type of Adam message to send to the task and is one of `set`, `get`, `obey`, `cancel`, `paramreply` or `syncrep`.

The format of the command to send a task an **obey** message is:

```
<taskname> obey <action> <command-line>
```

So, for example, we can instruct KAPPA to execute the add command with:

```
kappa obey add "in1=image1 in2=image2 out=result"
```

However, we don't receive any notification of when the obey has completed.

The action to be taken when a message of a particular type is sent by the task is specified by additional arguments consisting of a message type with a hyphen prepended followed by a Tcl command to be executed when the message is received. The command is executed in global scope in the same way as widget and X event bindings. So, for example:

```
kappa obey add "in1=image1 in2=image2 out=result" \
    -endmsg {puts "add finished"}
```

Will print "add finished" on the terminal when the obey has completed.

The various fields in the message can be inserted in the command to be executed wherever the percent character appears (similar to the substitutions performed when widget bindings are invoked). For example:

```
kappa obey stats "ndf=result" -inform {puts %V}
```

will print the messages output by the stats command on the terminal. The following tokens are recognised for all message types:

token	replaced by
`%C`	Message context
`%T`	Task name
`%N`	Message name
`%P`	Path
`%M`	Message id
`%S`	Status
`%V`	Message value
`%R`	Reply Token
`%%`	%

In the case of a **paramreq** message the value is in the form of a Tcl list and the following tokens are replaced by the appropriate element of the list:

token	replaced by
%n	parameter name
%p	prompt string
%d	parameter default
%h	help text
%e	error message

Dealing with a paramreq message is more complicated than any other message type because it demands that a reply is sent to the task before it will continue. The format of the command to send a **paramreply** message is:

```
<task-name> paramreply <reply-token> <reply>
```

The `<reply-token>` parameter is used by the message system to route the message back to the task that made the parameter request and can be extracted from the **paramreq** message.

About the simplest example is:

```
kappa obey cadd "in=image1 out=result" \
    -paramreq {kappa paramreply %R !}
```

This will reply with "!" whenever any parameter is prompted for. A more realistic example is:

```
proc prompt {task reptok param string default} {
#+
# Handles a prompt request from a task.
#-

# Create dialog box to display the prompt and receive the user's reply.
    toplevel .${task}_prompt
    wm title .${task}_prompt "Parameter prompt from ${task}"
    wm transient .${task}_prompt .

# Create the dialog box layout.
    label .${task}_prompt.label -bd 5 -text \
        "$task requires a value for $param"
    label .${task}_prompt.prompt -text $string -bd 5
    entry .${task}_prompt.entry -bd 2 -relief sunken
    .${task}_prompt.entry insert end $default
    button .${task}_prompt.ok -text OK -command "
        $task paramreply $reptok \[.${task}_prompt.entry get\]
        destroy .${task}_prompt
    "
    pack .${task}_prompt.label
    pack .${task}_prompt.ok -side bottom -pady 10
```

```
    pack .${task}_prompt.prompt .${task}_prompt.entry -side left
}
kappa obey cadd "in=image1 out=result" \
    -paramreq {kappa prompt %T %R %n %p %d}
```

Default handlers are provided for **paramreq**, **inform** and **sync** messages. The handler for **paramreq** messages is the one listed above. The handler for **inform** messages creates a dialog box (one for each task) containing a scrolling text widget and inserts the message into it. These ensure that, by default, important messages are not ignored; they are adequate during development but for production systems bespoke **inform** message handlers will almost certainly be required. The handler for **sync** messages calls the Tcl **update** command and then sends a **syncrep** message to the task.

If you want to ignore a message type then specify an empty string as the handler procedure. For example:

```
kappa obey stats "ndf=result logfile=stats.log" -inform ""
```

will run "stats" and log the output to a file instead of displaying them.

The format of the remaining command options that send messages to a task are:

cancel *action command-line*

set *parameter-name parameter-value*

get *parameter-name*

There are some additional command options that don't actually send any messages to the task:

kill kills the adam task (provided that the task was loaded by the `adamtask` procedure) and deletes the task procedure.

forget deletes the task procedure and removes the task from the list of known tasks so that it doesn't get killed when the Tcl application exits.

path returns 1 if a message system path to the task has been established and 0 if it hasn't. This can be used to wait for a task to be loaded before sending it any messages. For example:

```
adamtask kappa /star/bin/kappa/kappa_mon
set count 0
while {[kappa path] == 0} {
    after 100
    incr count
    if {$count > 100} {
       tk_dialog .loadError $title \
         "Timed out waiting for task kappa to start" error 0 OK
       kappa kill
    }
}
```

loads KAPPA and tries 100 times at 100 millisecond intervals to establish a message system path.

All the task commands return immediately the appropriate message has been sent; they do not wait for the task to acknowledge the receipt of the message. The following code illustrates how to wait for an action (in this case an obey) to be completed:

```
kappa obey stats -endmsg {set done 1}
tkwait variable done
```

Note that the set command is executed in global scope so in this case there is no need to declare **done** to be global. If **done** were to be set in a procedure as in:

```
procedure setdone {} {
     global done
     set done 1
}
kappa obey stats -endmsg setdone
```

then a global command is required. While the tkwait command is waiting, widget and X event bindings can still be triggered and adam messages received.

If a large number of adam messages are sent without waiting for an acknowledgement there is a risk of filling up the buffers in the message system and causing the system to deadlock. In order to avoid this happening you should:

- Call **update idletasks** to give the Tcl application the chance to respond to incoming adam messages.
- Use the mechanism described above to wait for acknowledgement messages whenever the task is expected to respond promptly.
- Use the command associated with the receipt of the acknowledgement to send the next message. For example, a series of **obey** messages can be 'chained' together by send each obey in response to the **endmsg** message from the previous **obey**.

4 Noticeboard Extension

The Adam notice board system (SUN/77) stores arrays of bytes in shared memory. Each notice board item has associated with it a data type and dimensions but the notice board system itself does not attach any meaning to them; it is up to the applications accessing the noticeboard to agree on the interpretation of the type and dimension information. The types recognizes by the Tcl/Tk interface to the noticeboard system and the way it interprets the data value are:

`_INTEGER` 4 byte signed integers.

`_REAL` 4 byte floating point numbers.

`_DOUBLE` 8 byte floating point numbers.

`_LOGICAL` 4 byte logical values where zero is FALSE and any other value is TRUE.

`_CHAR` A character string.

The notice board extension defines one new command **nbs**. The **nbs** command can be used to read nbs notice board items with:

```
nbs get <nbs_name>
```

which returns the values stored in the specified nbs item. If the type of the item is one of _INTEGER, _REAL, _DOUBLE or _LOGICAL the values stored in noticeboard are formatted into a list of integer, float or logical values as appropriate. If the item is of any other type the entire item is simply returned as a string with no interpretation.

Notice board items can be written with:

```
nbs put <nbs_name> <value>
```

where `<nbs_name>` is a fully qualified nbs item name and `<values>` is the values to be written. The format of `<values>` depends on the type of the nbs item; if it is _INTEGER, _REAL, _DOUBLE or _LOGICAL, it must be a list of values of that can be converted to the appropriate type by the usual rules and of exactly the correct length to fill the item. If the type is _CHAR it must by a string, no longer than the length of the item; the string will be padded with blanks to the correct length before being written to the noticeboard. If the item has any other type `<value>` must be a string of exactly the same length of the item.

The command:

```
nbs info <nbs_name>
```

returns a list of information about the nbs item. The first element of the list which indicates whether the item is a primitive item (1) or a structure item (0). If it is a structure item the remaining elements of the list are the names of the children of the item. If it is a primitive item the remaining elements are:

(1) The type.

(2) The length.

(3) A list of the dimensions.

Noticeboard items can also be "monitored"; at some specified time interval the value stored in the item is copied into a Tcl variable, again using the standard type conversions.

```
nbs monitor <nbs_name> <variable>
```

adds `<nbs_name>` to the list of NBS items to be monitored and associates it with the global Tcl variable `<variable>`.

```
nbs start <interval>
```

starts monitoring all the NBS items in the list with the interval between checks set to `<interval>` milliseconds.

```
nbs stop
```

stops the monitoring process and

```
nbs clear
```

empties the list of nbs items being monitored.

```
nbs monitor
```

returns a list of all the noticeboard items currently being monitored. Each element of the list consists of a two element list giving the notice board item name and the name of the corresponding Tcl variable name.

Examples are given in appendix B.

A Example gwm server script

```
#!/star/bin/awish
#
# gwm.tcl
#
# This file is an example of using the gwm widget and associated procedures
#
# It creates a gwm server window with scroll bars for scrolling the
# window and buttons for changing colours, printing, clearing and exiting.
# A crosshair is optionally displayed at the cursor position.
#
# Any arguments to the script that are not recognised as wish options
# are used as arguments for the widget creation command.
#

# Add the location of the gwm procedures to the auto load path.
    lappend auto_path /star/lib/startcl

# Create the gwm widget with scroll bars. The arguments to this script
# are concatonated with the command to create the widget and the resulting
# string evaluated so that the script arguments become additional arguments
# to the command.
    set create [concat {gwm_gwmWithScroll .gwm} $argv]
    set gwm [eval $create]

# Create and pack a frame for the control buttons.
    pack [frame .bottom] -side bottom -fill x
    pack [frame .buttons -relief sunken -border 2] -padx 3 -pady 3 \
        -side right -in .bottom

# Pack a label along side the button frame to match the appearance of the
# frame
    pack [label .bottom.fill -relief sunken -border 2 -anchor w] -fill both \
        -padx 3 -pady 3 -side left -expand y

# Create the command buttons.
    button .buttons.exit -text Exit -command exit -padx 10
    button .buttons.colours -text Colours -padx 10 \
        -command "gwm_colourDialog .col $gwm .buttons.colours"
    button .buttons.clear -text Clear -padx 10 -command "$gwm clear"
    button .buttons.ovclear -text "Clear Overlay" -padx 10 \
        -command "$gwm ovclear"
    button .buttons.print -text Print -padx 10 \
        -command "gwm_printDialog .pr $gwm .buttons.print"
    checkbutton .buttons.crosshair -text Crosshair -padx 10 \
        -variable crosshair -command crossHair

# Pack the buttons into the frame.
    pack .buttons.exit .buttons.clear .buttons.print .buttons.crosshair \
        -side left -expand y -padx 5 -pady 5
```

```
pack .buttons.colours -after .buttons.exit  -padx 3 -pady 3 \
            -side left -expand y

# Pack the "clear overlay" button if the window has an overlay.
    if [$gwm cget -overlay] {
        pack .buttons.ovclear -after .buttons.clear  -padx 3 -pady 3 \
            -side left -expand y
    }

# Bind a procedure that displays the current pointer position in a pop-up
# window to pressing mouse button 2.
    bind $gwm <ButtonPress-2> {showPointer %x %y}
    bind $gwm <ButtonRelease-2> {destroy .position}

# Change the cursor so that is doesn't interfere with the pop-up.
    $gwm configure -cursor draft_small

# Pack the gwm widget's frame into the top level widget. This is done last
# so that when the top level is resized it is the gwm widget that gets
# resized to fit rather than the frame containing the buttons.
    pack .gwm -in .

# Allow the window to be resized by the window manager by setting the
# minimum size.
    wm minsize . 1 1

# Map the window and find out how big it is and use this as the maximum
# size allowed by the window manager. Also constrain the minimum size so
# that the buttons are always visible.
    update idletasks
    wm maxsize . [winfo width .] [winfo height .]
    wm minsize . [winfo reqwidth .buttons ] [winfo reqwidth .buttons ]

# Change the main window title to be the name of gwm window
   wm title . [$gwm cget -gwmname]

proc crossHair {} {
#+
# This procedure is called whenever the "crosshair" checkbutton is toggled
# and either enables the crosshair whenever the pointer is in the gwm
# widget and binds the crosshair position to the pointer or disables the
# crosshair.
#-
    global crosshair
    global gwm
    if $crosshair {
        bind $gwm <Any-Motion> { %W set crosshair %x %y }
        bind $gwm <Any-Enter> { %W configure -crosshair yes }
        bind $gwm <Any-Leave> { %W configure -crosshair no }
    } {
        bind $gwm <Any-Motion> {}
        bind $gwm <Any-Enter> {}
        bind $gwm <Any-Leave> {}
        $gwm configure -crosshair no
```

```
    }
}

proc showPointer {x y} {
#+
# This procedure pops up a panel that displays values of the parameters
# x and y corrected for any scrolling of the gwm widget.
#-
    toplevel .position -bd 3 -relief raised
    wm overrideredirect .position 1
    global gwm
    set xpos [expr [winfo rootx $gwm] + $x]
    set ypos [expr [winfo rooty $gwm] + $y]
    wm geometry .position +$xpos+$ypos
    set x [expr $x - [$gwm cget -xoffset]]
    set y [expr $y - [$gwm cget -yoffset]]
    label .position.x -text "X = $x"
    label .position.y -text "Y = $y"
    pack .position.x .position.y
}
# End of script - enter event loop...
```

B Noticeboard Examples

B.1 Listing a noticeboard contents

```
proc showNbs {item} {
#+
# Lists the names and values of all the children of the noticeboard
# item $item. If $item is a noticeboard name, the entire noticeboard is
# listed.
#-
    set info [nbs info $item]
    if {[lindex $info 0]} {
        puts "$item: ([lindex $info 1]*[lindex $info 2]) \
            [lindex $info 3] [nbs get $item]"
    } {
        for {set i 1} {$i < [llength $info]} {incr i} {
           showNbs $item.[lindex $info $i]
        }
    }
}
```

C Low Level ADAM Message System Commands

The command:

```
adam_start name
```

initialises the Adam message system and registers *name* as the task name and creates the following additional commands:

- **adam_path** *name*

  Returns 1 if a path to *name* exists and 0 if it doesn't.

- **adam_receive**

  Receives an Adam message and returns a list who's elements are:

  command task msg_name path messid msg_status msg_value

  where *command* is one of:

  ```
  actstart paramreq paramrep inform sync syncrep trigger startmsg
  endmsg getresponse setresponse controlresponse
  ```

  If *command* is `paramreq` then `msg_value` is a list who's elements are:

  parameter_name prompt_string default_value help_text error_message

  Note that if there is no message available **adam_receive** will wait until a message arrives.

- **adam_send** ***task msg_name context msg_value***

  Sends a message to a task starting a new transaction. *context* must be one of:

  ```
  get set obey cancel control
  ```

  adam_send returns a list who's elements are the path and messid of the new transaction.

- **adam_reply** ***path messid msg_status msg_name msg_value***

  Sends a message on the transaction specified by *path* and *messid*. Valid values for *msg_status* are:

  ```
  actstart actcomplete paramrep paramreq inform sync syncrep trigger
  ```

- **adam_getreply** ***timeout path messid***

  Waits for an adam message to arrive on the given path and messid and returns a list whose elements are the same as the list returned by **adam_receive**.