CLM – A Language Binding for Common Lisp and OSF/Motif

User Guide and Reference Manual

Version 2.1

Project GINA

Andreas Bäcker
Christian Beilken
Thomas Berlage
Andreas Genau
Michael Spenke

GMD
(German National Research Center for Computer Science)
P.O. Box 1240
D-5205 Sankt Augustin 1
Federal Republic of Germany
e-mail: spenke@gmdzi.gmd.de

January 16, 1992

Copyright

Copyright 1992 GMD (German National Research Center for Computer Science)

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of GMD not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. GMD makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

GMD Disclaims All Warranties With Regard To This Software, Including All Implied Warranties Of Merchantability And Fitness, In No Event Shall GMD Be Liable For Any Special, Indirect Or Consequential Damages Or Any Damages Whatsoever Resulting From Loss Of Use, Data Or Profits, Whether In An Action Of Contract, Negligence Or Other Tortious Action, Arising Out Of Or In Connection With The Use Or Performance Of This Software.

Authors: Andreas Bäcker (baecker@gmdzi.gmd.de)
Thomas Berlage (berlage@gmdzi.gmd.de)
P.O. Box 1240
D-5205 Sankt Augustin 1
West Germany

Contents

Chapter 1

Introduction

CLM is a language binding for Common Lisp and OSF/Motif. It provides access to the functionality of the X toolkit intrinsics and the Motif convenience functions for Common Lisp. Client programs can use the OSF/Motif widget classes for their graphical user interface. Additionaly, client programs can use the CLX graphics functions to draw into Motif widgets[1], especially into widgets of classes `XmDrawingArea` and `XmDrawnButton`.

CLM consists of three components: A *CLM daemon*, a *CLM server*, and a package of Common Lisp functions. Both the CLM daemon and server are implemented in "C". The CLM daemon runs on an arbitrary machine in the network and listens for Lisp processes requesting to use CLM. The CLM daemon forks CLM server processes which communicate with their Lisp processes over a network-transparent TCP/IP connection. The CLM server offers the X toolkit and the Motif functionality as remote procedure calls. The package of Common Lisp functions provides a high-level interface to these remote procedures.

From the X server's point of view, the CLM server is an ordinary X client. The CLM server's functionality is to operate as a gateway between Motif and Common Lisp. The CLM functions send commands to the CLM server to create and modify widgets. The CLM server sends callbacks to the CLM application as a reaction to user actions suchs as moving a scrollbar's slider or pressing a push-button.

In the context of this manual, the term *CLM application* is a single (lightweight) process which runs inside a Lisp process and uses CLM[2]. The CLM daemon can serve an arbitrary number of Lisp processes. There may be an arbitrary number of concurrent CLM applications in a single Lisp process. The CLM daemon forks a CLM server process for each CLM application. This architecture protects concurrent processes from getting in trouble with the non-reentrant Motif code. Figure 1 shows the interaction between Lisp, the CLM daemon and server and the X server.

Lisp processes must not necessarily run on the same host as the CLM daemon. This allows CLM applications to run on hardware which is not capable of running X and Motif. CLM automatically chooses the right communication media for local and remote Lisp processes.

The CLM server solution is very efficient and results in good performance at the user interface level. Redraw operations and geometry management operations are performed

[1] This functionality is only available when the CLX package was loaded before compiling CLM

[2] The *multiprocessing facility* is an extension to Common Lisp and may not be available in all Lisp systems. In this case, it is only possible to run one CLM application per Lisp process.

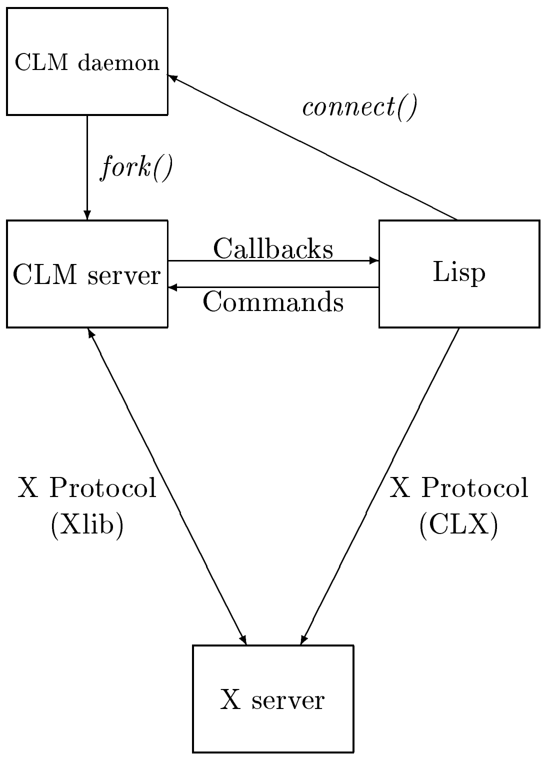

Figure 1.1: Communication between the CLM server, the X server and Lisp

by the X Toolkit Intrinsics and the Motif widgets and are handled locally in the CLM server. User interaction like browsing through menu items is also handled by the Motif widgets and requires only the execution of C code. Interactions like clicking a push-button lead to the execution of callbacks and cause an interaction with Lisp. Communication time is typically small and doesn't cause any substantial delays.

Chapter 2

Getting Started

To use CLM, programmers must carry out the following three steps:

1. Start a CLM daemon.
2. Load the CLM package.
3. Load their CLM application(s).

2.1 The CLM Daemon

Before running any CLM application, at least one CLM daemon must be started. A single CLM daemon can serve an arbitrary number of Lisp processes, but there may be any number of CLM daemons in a network. A CLM application may connect to any CLM daemon in the network. Multiple concurrent CLM applications in a single lisp process may choose to connect to different CLM daemons.

If you want two CLM daemons to run on the same machine, you must specify different TCP ports for them. Furthermore, all applications that connect to a specific CLM deamon run their interface under the user ID of the daemon. In particular, they use the X resource files (`.Xdefaults`) of this user.

If it is supported by the Lisp system and if Motif is available on the local machine, CLM server processes can be directly created from inside Lisp when necessary. This is the default configuration for CLM Version 2.1, so you normally do not need to start the CLM daemon as in previous versions.

2.1.1 Starting the CLM Daemon from a Command Interpreter

Running the CLM daemon from a command interpreter is useful if a single daemon should serve multiple Lisp processes on a single host or on multiple remote hosts. It is also useful if the Lisp process runs on a remote machine which is not capable of running UNIX, X and Motif.

To start the CLM daemon, it is necessary to have a running X server because the CLM daemon opens an X window which displays a list of active CLM server processes.

In a properly installed CLM system, it is sufficient to run the command

```
clmd
```

from a command interpreter. This script should either be in the current working directory or should be accessible through the search path. The script runs a shell script which starts the CLM daemon and supplies it with all necessary arguments. The `clmd` script also takes the command line options which are defined by the X Toolkit Intrinsics and passes them to the CLM daemon (e.g., `-geometry`, `-name`).

2.1.2 The CLM Daemons Window

The CLM daemon's window contains a *Quit* button, a *Kill* button and a list of active CLM server processes. The *Quit* button is used to terminate the CLM daemon. All CLM server child processes keep on running after the CLM daemon has terminated. The *Kill* button is used to kill all CLM server processes which are selected in the list of processes.

For each active CLM server, the CLM daemon displays some information about the type of connection which is used for the TCP/IP protocol and the CLM server's process ID (*pid*).

2.2 The CLM Package

The CLM package contains the Common Lisp part of CLM. It consists of a (compiled) lisp module and an optional set of object files which contain foreign functions. CLM applications which use functions or variables from the CLM package must either import the CLM package into their source code by a `(use-package :xtk)` statement or must use the prefix `xtk:` for each exported symbol in the CLM package.

2.2.1 Loading the CLM Package

To load the CLM package into a Lisp world after it has been installed, the following steps are necessary:

- Optionally load the CLX package. This is not really required for CLM. If the CLX package is not loaded, the functionality of CLM which deals with CLX windows and graphics will not be available. This functionality of CLM will still not be available if the CLX package is loaded after the CLM package has been loaded. This functionality will also not be available if the CLX package wasn't loaded when the CLM package was compiled.
- Change to the directory where the CLM package is stored.
- Load the CLM package and optional foreign functions by calling `(load "clm")`. These commands load the CLM package and the foreign function modules.
- Optionally save the lisp image. The CLM package contains the non-exported function (`xtk::save-lisp` *image-name*) which can be used to save the Lisp image. This function may not be available on all Lisp systems.

2.3 Troubleshooting

- When the CLM daemon, server or the CLM application fails to open a display connection, make sure that your applications are allowed to connect to the X server. Server access is enabled with the `xhost` program.
- If the CLM daemon fails to come up, it may be necessary to remove the file `/tmp/clm_socket`.
- If the windows of the CLM daemon or your application don't appear on the screen, check the name of the display host you're using.

Chapter 3

CLM Variables

The variables described in this chapter can be used to configure the default behavior of CLM. Table 3.1 gives an overview on the variables and their semantics.

Variable	Description
`xtk::*debug-mode*`	Enables/disables synchronous CLX requests
`xtk::*default-display-host*`	Host where the X display should be opened
`xtk::*default-display-number*`	Default number of the X display
`xtk::*default-screen-number*`	Default number of the X screen
`xtk::*default-server-host*`	Host where the CLM daemon is running
`xtk::*clm-binary-directory*`	Directory where server binary resides
`xtk::*clm-binary-name*`	Name of the server binary
`xtk::*xt-tcp-port*`	TCP port number usedby the daemon
`xtk::*motif-connection*`	CLM server connection (per CLM application)
`xtk::*x-display*`	CLX display connection (per CLM application)

Table 3.1: CLM variables

`*debug-mode*` (Variable)

Description: The value of this boolean variable is used as the default value for the `:sync-clx` keyword argument of `run-motif-application` (see page 4.6). The variable's default value is nil.

If the variable's value is non-nil, all CLX display connections are opened in a synchronous mode. This simplifies debugging of calls to CLX functions, but in general slows down execution of calls to CLX functions.

`*default-display-host*` (Variable)

Description: This variable contains the name of the host where the X display should be opened. The default value is `"localhost"`, which addresses the X display of the host where the CLM server is running. The value of this variable is used as the default value for the `:display-host` argument of the function `run-motif-application`.

If the CLM server runs on a remote host, `*default-display-host*` addresses a host relative to the remote host. In this case, the display host name `"localhost"` should not be used because it addresses the remote host's display. Use the name of your local host explicitly in this case.

Example:

```
> (setf xtk::*default-display-host* "olymp")
"olymp"

> (setf xtk::*default-display-host*
        (or (eq xtk::*default-server-host* "localhost") "olymp"))
"olymp"

>
```

`*default-display-number*` (Variable)

Description: This variable contains the number of the X display that should be opened. It's default value is `0`. The value of this variable is used as the default value for the `:display-number` argument of the function `run-motif-application`.

Example:

```
> (setf xtk::*default-display-number* 1)
1

>
```

`*default-screen-number*` (Variable)

Description: This variable contains the number of the X screen that should be opened. It's default value is `0`. The value of this variable is used as the default value for the `:screen-number` argument of the function `run-motif-application`.

Example:

```
> (setf xtk::*default-screen-number* 1)
1

>
```

`*default-server-host*` (Variable)

Description: This variable contains the name of the host where the CLM server is running. It's default value is `nil`, which starts the CLM server directly from Lisp. Use the value `"localhost"` to use the CLM daemon running on the local host. The value of this variable is used as the default value for the `:server-host` argument of the function `run-motif-application`.

Example:

```
> (setf xtk::*default-server-host* "olymp")
"olymp"

> (progn
    ;; Run the CLM server on the remote host "olymp".
    ;; Use the local host's display.
    ;; (Remote = "olymp" Local = "zeus")
    (setf xtk::*default-server-host* "olymp")
    (setf xtk::*default-display-host* "zeus"))
"zeus"

>
```

`*clm-binary-directory*` (Variable)

Description: This variable contains the name of a directory where the CLM server binary can be found. The value may either be a string or a list of strings. All the directories are searched in the given sequence for the binary. This variable is only used when `*default-server-host*` is `nil`.

`*clm-binary-name*` (Variable)

Description: This variable contains the name of the CLM server executable. The default value is `"clm-server"`.

`*xt-tcp-port*` (Variable)

Description: This variable contains the number of the TCP port that the CLM daemon uses. The default value is 7000. This variable needs to be changed if you want to start multiple CLM daemons on the same machine. These daemons must have different port numbers. This variable determines which port the CLM application contacts.

`*motif-connection*` (Variable)

Description: This variable references the TCP/IP connection to the CLM server. Each CLM application has its own instance of this variable. Normally, there is no need for an application to use this variable.

`*x-display*` (Variable)

Description: This variable references the application's CLX display connection. This connection is automatically opened if the `:use-clx` keyword argument of the `run-motif-application` is given a non-nil value. This variable can be used for the *display* argument of the CLX functions. Each CLM application has its own copy of this variable.

Note: This variable is only available if the `xlib` package was present in the Lisp image at the time the CLM code was loaded. This variable is `nil` if the `:use-clx` keyword argument of `run-motif-application` is `nil`.

Chapter 4

Applications

A CLM application is startet using the `run-motif-application` function. Examples are discussed in detail in section 4.7.1, page 19.

In general, only the `init-function` argument of `run-motif-application` must be given, which is the name of a function that initializes the application and creates the application's widget hierarchy. The function `run-motif-application` automatically establishes a connection to the CLM server, calls the `init-function` and then calls `app-main-loop` to run the application main loop. The application main loop receives all incoming callbacks, X events and timer callbacks and handles them by calling the handler functions that were registered by the application. It is convenient to encapsulate the call to `run-motif-application` in an additional function. This function defines the external interface of the CLM application and calls `run-motif-application` with all necessary arguments.

4.1 CLM Applications in a Distributed Environment

CLM applications can specify a remote host where the Lisp system can find a CLM daemon. The keyword argument `:server-host` of `run-motif-application` is used to tell a CLM application the name of a host where it can find the a running CLM daemon. The default value of `:server-host` is taken from the variable `xtk::*default-server-host*`. The variable's default value is `nil`, which tells CLM not to use the CLM daemon.

The keyword argument `:display-host` of `run-motif-application` is used to tell a CLM application where it's associated CLM server process should open the X display. The argument's default value is taken from the variable `xtk::*default-display-host*`. The variable's default value is `nil`, which starts the CLM server directly from Lisp.

CLM applications can also select a display on the display host by setting the `:display-number` argument of `run-motif-application`. The default display number is taken from the variable `xtk::*default-display-number*`. The variable's default value is 0.

The default values of the keyword argument mentioned above need not to be changed if the Lisp process, the CLM daemon and the X server all run on the same host.

4.2 Running Multiple CLM Applications

CLM applications can run in an extra (lightweight) Lisp process. This allows multiple independent CLM applications to run in a single Lisp process. To run an application in an extra process, the `:extra-process` keyword argument must be given a non-nil value. The `:process-name` keyword argument then names the process.

CLM applications can access their CLM connection and X display connection by using the special variables `xtk::*motif-connection*` and `xtk::*x-display*`. Each CLM application runs in an environment where these variables are local to the application's process.

4.3 Using CLX in a CLM Application

If the `:use-clx` keyword argument is set to a non-nil value, an additional CLX display connection is opened on the display specified by the `:display-host` and `:display-number` arguments. This allows applications to use the CLX graphics routines for drawing into Motif widgets, especially into widgets of classes `XmDrawingArea` and `XmDrawnButton`. The CLM function `make-clx-window` can be used to create a CLX window structure from a Motif widget ID which is suitable for use with any CLX function.

`make-clx-window` (Function)

Syntax: `(defun make-clx-window (widget))`

Description: This function creates an CLX window structure from a given Motif widget ID. This window structure may be used with any CLX function. The widget must be realized, otherwise it does not have an associated window yet.

Parameters: *widget* Specifies the Motif widget ID.

Example:

```
> (setf clx-window (make-clx-window widget))

.....
> (setf (xlib:window-event-mask clx-window)
        (xlib:make-event-mask :focus-change :structure-notify))

.....
```

4.4 The Application Main Loop

The application main loop receives all incoming callbacks, X events, timer callbacks and protocol callbacks and dispatches them by calling the handler functions which were registered by the CLM application. Normally, the application main loop is called automatically by `run-motif-application`, so there is no need for a CLM application to call the application main loop explicitely.

CLM applications may set the `:execute-main-loop` argument to `nil` and call `app-main-loop` in their init function. This is sometimes useful when the application needs some dynamically bound global variables which are usually initialized in a `let` statement. The `app-main-loop` function can then be called from inside the `let` statement, so that all callback handlers and event handlers are executed inside this lexical environment. See the `xclock` program in secxtion 4.7.1 for an example.

CLM applications can specify a function to be called by `run-motif-application` after the application main loop has terminated. This function usually does cleanup actions like closing files etc. The after-function is called even in case the application terminates abnormally, for example by an abort. The after-function is executed in the lexical environment of `run-motif-application`, but outside the lexical environment of the init function. The after function is specified by the `:after-function` keyword argument. Additional arguments which are passed to the after-function may be specified by the `:after-function-args` keyword argument.

4.5 Terminating an Application

If the Motif window manager mwm is running, the application main loop is automatically terminated when the *Close* function of mwm is triggered (usually by double-clicking the system menu button of the mwm decorations).

If the Motif window manager is not running, applications must provide a way to terminate themselves. This may be realized by creating a *Quit* Button somewhere in the application's menu. The application can attach the predefined callback handler `quit-application` to this button. When executed, `quit-application` destroys the widget hierarchy and terminates the application main loop. Applications can use the function `is-mwm-running` to test whether mwm is currently running.

4.6 Debugging Motif Applications

For debugging purposes, the CLX connection can be put in a synchronous mode by passing a non-nil value to the `:sync-clx` keyword argument.

When errors occur in the init function or in the main loop, CLM installs an error handler that prints out the error and does not fall into the debugger. In the main loop, execution continues in the loop. In the init phase, the after function and cleanup actions are executed and an extra process exits cleanly. This avoids handling multiple processes in the debugger because interactive applications are usually restartable.

To enter the debugger, you can supply the keyword `:just-print-errors` to `run-motif-application` as `nil`. Alternatively, you can supply your own error handler, e.g. to print a complete backtrace in addition to the error message. If your error handler returns, the debugger is entered. If the error handler wants to resume execution, it must throw to `xtk:proceed-from-error`.

From the main loop, CLM sets up a restart choice for the debugger to restart the main loop. This choice will be presented by the debugger in a system-dependent way.

`run-motif-application` (Function)

Syntax:

```
(defun run-motif-application (init-function &key
                              (init-arguments nil)
                              (application-name "clm")
                              (application-class "Clm")
                              (server-host *default-server-host*)
                              (display-host *default-display-host*)
                              (display-number *default-display-number*)
                              (screen-number *default-screen-number*)
                              #+xlib (use-clx nil)
                              (extra-process t)
                              (process-name "Motif application")
                              (execute-main-loop t)
                              (sync-clx *debug-mode*)
                              (after-function nil)
                              (after-arguments nil)
                              (error-handler 'default-error-handler)
                              (just-print-errors t)))
```

Description: This function is used to run a CLM application. It establishes a connection to the CLM server on the host specified by the `:server-host` argument. If the connection is opened successfully, it calls the function specified by the `init-function` argument. The `init-function` is applied to the elements of the `:init-arguments` argument. After calling the init function and if the `:execute-main-loop` argument is `t`, the application main loop function is called. When the application main loop has terminated, the function specified by the `:after-function` argument is called with an argument list containing the elements of the `:after-function-args` argument. The after function is called even in the case when the init function or the application main loop terminate abnormally. After executing the after function, the connection to the CLM server and the optional CLX connection are closed automatically.

Parameters:

init-function Specifies the name of the function that initializes the Motif application. This function normally creates the widget hierarchy, attaches callbacks to the widgets and creates all application specific data structures.

init-arguments Specifies a list of arguments to the init function.

application-name Specifies the application's name as a string. This name can be used as a prefix for resource specifications in the application's default file.

application-class Specifies the application's class as a string. This class can be used as a prefix for resource specifications in the application's default file.

server-host Specifies the host where the CLM daemon is running as a string. The value `nil` can be used to start the server directly from Lisp.

display-host Specifies the host where the CLM application opens the X display. All windows corresponding to the widgets of this application are created in the X server of this host.

display-number Specifies the number of the display that should be opened by the CLM server.

screen-number Specifies the number of the screen that should be opened by the CLM server.

use-clx Specifies whether an extra CLX display connection should be opened. If the value is non-nil, a connection is established to the display specified by the `:display-host` and `:display-number` arguments. The CLM application can access the CLX display structure by using the exported variable `*x-display*`.

execute-main-loop Specifies whether the application main loop is executed automatically. If the value is non-nil the main loop is started automatically. Otherwise, the init function is responsible for calling `app-main-loop`. Then the call to `app-main-loop` usually is the last expression in the init function.

extra-process Specifies whether the CLM application should run in an extra lisp process. If the value is non-nil, an extra lisp process is started which executes the init-function, the application main loop and the after function.

process-name Specifies the name of the process if the CLM application is running in an extra lisp process.

sync-clx Specifies whether the optional CLX display connection should be synchronous. This allows easier debugging, but slows down application speed.

after-function Specifies the function to be called after the application main loop has terminated. The after function is called even in case of an abort.

after-arguments Specifies a list of arguments to the after function.

error-handler Specifies a function as an error handler. The error condition is passed as the parameter to this function. The function may return to enter the debugger, or it throws to `xtk:proceed-from-error` to resume execution.

just-print-errors If this parameter is `t`, the default error handler just prints out the error message and then resumes execution. If it is `nil`, the debugger is entered when errors occur.

Example: A detailed example is discussed at the end of this chapter.

`app-main-loop` (Function)

Syntax: `(defun app-main-loop ())`

Description: This function realizes the main loop of a CLM application. It receives all incoming callbacks, X events and timer callbacks and handles them by calling the handler functions which were registered by the application.

The application main loop is called automatically by the function `run-motif-application` when the `:execute-main-loop` argument of `run-motif-application` has a non-nil value. If this argument is `nil`, then the application's init-function is responsible for calling `app-main-loop`.

If the application main loop is terminated abnormally, the error handler is invoked, which may restart the main loop.

`quit-application` (Function)

Syntax: `(defun quit-application (widget client-data &rest call-data))`

Description: This function is a predefined callback handler. It is used to destroy the application's shell widget and to terminate the application main loop. The `client-data` argument must contain the widget ID of the application's shell widget, which is returned by the function `create-application-shell`. The other arguments are not used.

Note: This callback handler is automatically called when the *Close* command from the Motif window manager's system menu is executed. If mwm is not running, CLM applications must call `quit-application` in order to terminate themselves. Applications can test whether mwm is running by calling `is-mwm-running`.

widget	Not used.
client-data	Must contain the widget ID of the application's shell widget.
call-data	Not used.

`is-mwm-running` (Function)

Syntax: `(defun is-mwm-running (shell))`

Description: This function returns `t` if the Motif window manager mwm is running on the shell's display and `nil` otherwise.

shell	Specifies the shell widget ID.

4.7 Examples

In this section, two Motif applications which use most of the features supplied by `run-motif-application` are described in detail. The first application is a simple digital clock that updates its display every second using the timer mechanism supplied by CLM. The second one monitors the (lightweight) processes currently running in the Lisp process.

4.7.1 The Application `xclock`

The `xclock` application (see programm 1, page 20) is a simplified version of the *xclock* application coming with the X window system. It displays the current time in digital format and is updated every second. An update interval may be specified by the optional `update-interval` argument.

The timer callback function needs the ID of the label widget to be able to display the new time. This is achieved by creating a lexical environment in which the variable `*clock-widget*` is bound to the label's widget ID. The timer callback function must be executed inside this environment. Therefore, the application main loop must be called from inside the `let` statement in the function `init-xclock`. This is realized by setting the `:execute-main-loop` argument to `nil` and then calling `app-main-loop` as the last form inside the `let` statement.

Note: The label's widget ID could also be passed to the timer callback function by using the client data mechanism. In general, CLM application would prefer this mechanism of passing data to callback functions over using global variables.

4.7.2 The Application `xprocesses`

The `xprocesses` application (see program 2, page 21) displays the names of all running processes in a Motif list widget. The list of processes is refreshed periodically. An initial update interval may be specified by the optional `update-interval` argument. A Motif scale widget can be used to change the update interval.

The application's structure is similar to the `xclock` application. One main difference is the mechanism how data structures are passed to the timer callback function. In both examples, the timer callback function needs a widget ID to display it's data. In the `xclock` example, this is achieved by dynamically binding a global variable and executing the application main loop in the scope of this binding. In the `xprocesses` application, the widget ID is passed as the `client-data` argument to the timer callback function.

```
(in-package 'xclock)
(use-package 'xtk)

'(xclock::xclock)   ;; evaluate this to run the xclock application

(defun xclock (&optional (update-interval 1000))
  "Run the xclock application"
  (run-motif-application 'init-xclock
                         :init-arguments (list update-interval)
                         :execute-main-loop nil
                         :application-name "clock"
                         :application-class "Clock"
                         :process-name "Motif Clock Application"))

(defun init-xclock (update-interval &aux shell)
  "Create the widget hierarchy for the clock application"
  ;; Create the application's main window
  (setf shell (create-application-shell :allow-shell-resize t))
  (let ((*clock-widget* (create-label shell "ClockLabel"))
        (timer (create-timer update-interval 'tick nil :active-in-handler t)))
    ;; Make *clock-widget* globally available so the
    ;; timer callback can access the label widget
    (declare (special *clock-widget*))
    ;; Initialize clock
    (tick timer nil)
    ;; Realize widget hierarchy
    (realize-widget shell)
    ;; Enter application main loop
    (app-main-loop)))

(defun tick (timer client-data &aux now)
  "one time interval is over => refresh the clock"
  (declare (special *clock-widget*)
           (ignore timer client-data))
  (setq now (multiple-value-list (decode-universal-time (get-universal-time))))
  ;; Display new time in the label widget
  (xtk::set-values *clock-widget* :label-string
                   (format nil "~2,'0d:~2,'0d:~2,'0d"
                           (third now) (second now) (first now))))
```

Programm 1: The application `xclock`

```
(in-package 'xprocesses)
(use-package 'xtk)

;; Evaluate this to start the xprocesses application
'(xprocesses::xprocesses 999)

(defun xprocesses (&optional (update-interval 5000))
  "Run the xprocesses application"
  (run-motif-application 'init-processes
                         :init-arguments (list update-interval)
                         :application-name "xprocesses"
                         :application-class "Xprocesses"
                         :process-name "Process Monitor"))

(defun init-processes (update-interval &aux shell row list-w scale timer)
  "Create the widget hierarchy for the xprocesses application"
  ;; Create the application's main window
  (setf shell (create-application-shell :allow-shell-resize t))
  ;; Create a row-colum widget which contains the list widget and a scale
  (setf row (create-row-column shell "Row"))
  ;; Create a list widget for displaying the processes
  (setf list-w (create-list row "ProcessList" :visible-item-count 5))
  (setf scale (create-scale row "Scale" :orientation :horizontal
                            :show-value t :value update-interval
                            :minimum 500 :maximum 5000))
  (setf timer (create-timer update-interval 'display-processes list-w))
  ;; Add callback to adjust the timer interval
  (add-callback scale :value-changed 'set-new-interval timer)
  ;; Initialize contents of the list widget
  (display-processes timer list-w)
  ;; Realize widget hierarchy
  (realize-widget shell))

(defun set-new-interval (scale timer call-data)
  ;; Scale's call-data is the new slider position
  (declare (ignore scale))
  (change-timer timer call-data))

(defun display-processes (timer list-w &aux ap)
  "Display all processes in the list widget"
  (declare (ignore timer))
  (setq ap (all-processes))
  (set-values list-w :visible-item-count (length ap))
  (apply 'set-items (cons list-w ap)))

(defun all-processes ()
  "Return a list of strings describing the processes"
  (or #+excl (mapcar 'mp::process-name mp::*all-processes*)
      #+lucid (mapcar 'process-name *all-processes*)
      '("Not implemented")))
```

Programm 2: The application `xprocesses`

Chapter 5

Widgets

The functions described in this chapter are used to create and manipulate the widget hierachy of a CLM application. Widgets are referenced to by *widget IDs*. The functions which create widgets return a widget ID which can later be used as arguments to other functions to identify the widget on which a certain operation is to be performed.

Most of the functions described in this chapter have the same semantics as their corresponding function in the X Toolkit Intrinsics. Refer to X Toolkit Intrinsic's documentation for a more detaild description. Tables 5.1, 5.2 and 5.3 show the CLM functions described in this chapter and their corresponding "C" functions.

CLM function	OSF/Motif function
`create-application-shell`	`XtAppCreateShell()`

`create-widget`
`create-unmanaged-widget`
`create-named-widget`
	`XtCreateWidget()`
	`XtManageChild()`

`destroy-widget`	`XtDestroyWidget()`
`realize-widget`	`XtRealizeWidget()`
`manage-widgets`	`XtManageChildren()`
`unmanage-widgets`	`XtUnmanageChildren()`
`map-widgets`	`XtSetMappedWhenManaged()`
`unmap-widgets`	`XtSetMappedWhenManaged()`
`set-sensitive`	`XtSetSensitivity()`
`set-insensitive`	`XtSetSensitivity()`
`get-parent`	`XtParent()`
`make-clx-window`	*none*
`xt-translate-coordinates`	`XtTranslateCoordinates()`

Table 5.1: Widget functions

5.1 Creating Widgets

The root of the widget tree is created with the `create-application-shell` function. Child widgets are created using either the `create-...` functions shown in tables 5.2 and 5.3 or one of the functions `create-widget`, `create-unmanaged-widget` and `create-named-widget`[1].

5.1.1 Using the `create-...` Functions

CLM provides a create function for each Motif widget and gadget class. Except for the function `create-application-shell` all these functions have an identical argument list. Their first argument is always the parent's widget ID. The second argument is a string which is used to name the widget. This name can be used in resource specifications in the application's defaults file. An arbitrary number of resources may be specified after the second argument. Resources are given as an alternating sequence of resource names and values.

CLM automatically manages widgets of most classes when they are created. The last column of table 5.2 and 5.3 shows which classes are automatically managed. An asterisk in this column indicates that the widget is initially *unmanaged*. This applies to all dialog classes and for shell classes. For most applications, the default initial managed state of a widget is sufficient. Clients may override this default state by supplying the resource `:managed` in a widget's resource list with a resource value of either `t` or `nil`.

The three CLM functions `create-toplevel-shell`, `create-override-shell` and `create-transient-shell` call `XtCreatePopupShell()` with the appropriate widget class pointer. Dialogs which use one of these functions to create their shell widget must be popped up using the function `popup` with the shell widget ID as it's first argument. Dialogs which use an `XmDialogShell` for their shell widget must have an *unmanaged* `XmBulletinBoard` (or subclass) widget as their only direct child and must be popped up by calling the function `manage-widgets` on the `XmBulletinBoard` widget. Dialogs which are created by one of the `create-...-dialog` functions must be popped up using `manage-widgets` on the widget ID returned by the `create-...-dialog` function.

5.1.2 Using the `create-widget` Function

The functions `create-widget`, `create-unmanaged-widget` and `create-named-widget` are used to create widgets of an arbitrary class. The widget's class is specified by a keyword symbol, which can be derived from the Motif widget class name as follows: Remove the prefixes `Xm` and `Xt` and the suffix `WidgetClass`. Insert a hyphen before all upper case letters except before the first one. Substitute all upper case letters with the corresponding lower case letter. Example: The Motif widget class `XmToggleButtonWidgetClass` is specified by the keyword `:toggle-button`. Table 5.4 gives an overview of the supported widget classes. The shell classes described in this table must be created with the function `create-popup-shell`, which is described in section 5.2.

The CLM functions `create-widget` and `create-unmanged-widget` actually call `create-named-widget` to create the widget. The function `create-widget` creates a man-

[1] The functions `create-widget`, `create-unmanaged-widget` and `create-named-widget` may disappear in future releases. Programmers should use the `create-...` functions.

CLM function	*C* function	unmanaged
create-arrow-button	XmCreateArrowButton()	
create-arrow-button-gadget	XmCreateArrowButtonGadget()	
create-bulletin-board	XmCreateBulletinBoard()	
create-bulletin-board-dialog	XmCreateBulletinBoardDialog()	*
create-cascade-button	XmCreateCascadeButton()	
create-cascade-button-gadget	XmCreateCascadeButtonGadget()	
create-command	XmCreateCommand()	
create-dialog-shell	XmCreateDialogShell()	
create-drawing-area	XmCreateDrawingArea()	
create-drawn-button	XmCreateDrawnButton()	
create-error-dialog	XmCreateErrorDialog()	*
create-file-selection-box	XmCreateFileSelectionBox()	
create-file-selection-dialog	XmCreateFileSelectionDialog()	*
create-form	XmCreateForm()	
create-form-dialog	XmCreateFormDialog()	*
create-frame	XmCreateFrame()	
create-information-dialog	XmCreateInformationDialog()	*
create-label	XmCreateLabel()	
create-label-gadget	XmCreateLabelGadget()	
create-list	XmCreateList()	
create-main-window	XmCreateMainWindow()	
create-menu-bar	XmCreateMenuBar()	
create-menu-shell	XmCreateMenuShell()	
create-message-box	XmCreateMessageBox()	
create-message-dialog	XmCreateMessageDialog()	*
create-override-shell	XtCreatePopupShell()	
create-option-menu	XmCreateOptionMenu()	
create-paned-window	XmCreatePanedWindow()	
create-popup-menu	XmCreatePopupMenu()	
create-prompt-dialog	XmCreatePromptDialog()	*
create-pulldown-menu	XmCreatePulldownMenu()	
create-push-button	XmCreatePushButton()	
create-push-button-gadget	XmCreatePushButtonGadget()	
create-question-dialog	XmCreateQuestionDialog()	*
create-radio-box	XmCreateRadioBox()	
create-row-column	XmCreateRowColumn()	
create-scale	XmCreateScale()	
create-scroll-bar	XmCreateScrollBar()	

Table 5.2: Widget creation functions (1)

CLM function	*C* function	unmanaged
`create-scrolled-list`	`XmCreateScrolledList()`	
`create-scrolled-text`	`XmCreateScrolledText()`	
`create-scrolled-window`	`XmCreateScrolledWindow()`	
`create-selection-box`	`XmCreateSelectionBox()`	
`create-selection-dialog`	`XmCreateSelectionDialog()`	*
`create-separator`	`XmCreateSeparator()`	
`create-separator-gadget`	`XmCreateSeparatorGadget()`	
`create-text`	`XmCreateText()`	
`create-text-field`	`XmCreateTextField()`	
`create-toggle-button`	`XmCreateToggleButton()`	
`create-toggle-button-gadget`	`XmCreateToggleButtonGadget()`	
`create-toplevel-shell`	`XtCreatePopupShell()`	
`create-transient-shell`	`XtCreatePopupShell()`	
`create-warning-dialog`	`XmCreateWarningDialog()`	*
`create-work-area`	`XmCreateWorkArea()`	
`create-working-dialog`	`XmCreateWorkingDialog()`	*

Table 5.3: Widget creation functions (2)

aged widget whose name is the symbol-name of the widget class keyword with all letters converted to lower case. The function `create-unmanaged-widget` is equivalent to `create-widget`, except that the widget is created unmanaged.

An arbitrary number of widget resources may be set when creating a widget. The widget's resources are supplied by the `&rest` argument `resources`. They are given as an alternating sequence of resource names and their values.

`create-application-shell` (Function)

Syntax: `(defun create-application-shell (&rest resources))`

Description: This function is used to create a CLM application's root window. It creates a widget of class `applicationShellWidgetClass` and returns it's widget ID. The key `:name` is recognized in the resource list to set the name of the application shell. This should usually be identical to the name of the complete application.

The function attaches a protocol callback (see chapter 6) to the shell widget, which executes the predefined callback handler `quit-application`. This callback is triggered upon execution of the *Close* function of the Motif window manager `mwm`. The protocol callback can be removed using the function `remove-protocol-callback`.

Note: This function must only be used to create the application's primary shell widget. Application which have one primary window should use

CLM widget class	Motif widget class
:dialog-shell	xmDialogShellWidgetClass
:menu-shell	xmMenuShellWidgetClass
:override-shell	overrideShellWidgetClass
:toplevel-shell	topLevelShellWidgetClass
:transient-shell	transientShellWidgetClass
:arrow-button-gadget	xmArrowButtonGadgetClass
:arrow-button	xmArrowButtonWidgetClass
:bulletin-board	xmBulletinBoardWidgetClass
:cascade-button-gadget	xmCascadeButtonGadgetClass
:cascade-button	xmCascadeButtonWidgetClass
:command	xmCommandWidgetClass
:drawing-area	xmDrawingAreaWidgetClass
:drawn-button	xmDrawnButtonWidgetClass
:file-selection-box	xmFileSelectionBoxWidgetClass
:form	xmFormWidgetClass
:frame	xmFrameWidgetClass
:gina-view	xmDrawingAreaWidgetClass
:label-gadget	xmLabelGadgetClass
:label	xmLabelWidgetClass
:list	xmListWidgetClass
:main-window	xmMainWindowWidgetClass
:message-box	xmMessageBoxWidgetClass
:paned-window	xmPanedWindowWidgetClass
:push-button-gadget	xmPushButtonGadgetClass
:push-button	xmPushButtonWidgetClass
:row-column	xmRowColumnWidgetClass
:scale	xmScaleWidgetClass
:scroll-bar	xmScrollBarWidgetClass
:scrolled-window	xmScrolledWindowWidgetClass
:selection-box	xmSelectionBoxWidgetClass
:separator-gadget	xmSeparatorGadgetClass
:separator	xmSeparatorWidgetClass
:text-field	xmTextFieldWidgetClass
:text	xmTextWidgetClass
:toggle-button-gadget	xmToggleButtonGadgetClass
:toggle-button	xmToggleButtonWidgetClass

Table 5.4: Widget classes of CLM

the ApplicationShell widget for the root of their widget hierarchy and must call `realize-widget` on the ApplicationShell's widget ID.

CLM applications which need more than one primary window (see example below) should use TopLevelShells which are created as popup children (using either the function `create-popup-shell` or the function `create-top-level-shell`) of the ApplicationShell widget. The application must then call the function `popup` on each primary TopLevelShell in order to display it. The application must not call the function `realize-widget` on the ApplicationShell.

Parameters: *resources* An arbitrary number of resource/value pairs.

Example:

```
> (defun hello-world (&aux shell)
    (setq shell (create-application-shell))
    (create-label shell "label" :label-string "Hello, world!")
    (realize-widget shell))
HELLO-WORLD

> (run-motif-application 'hello-world)
```

Example:

```
> (defun two-primary-windows ()
   (let* ((shell (create-application-shell))
          ;; Use create-popup-shell for the first shell widget
          (top-1 (create-popup-shell "top1" :toplevel-shell shell))
          (button-1 (create-widget :push-button top-1
                    :label-string "Popup-1"))
          ;; Use create-toplevel-shell for the second shell
          (top-2 (create-toplevel-shell shell "top2"))
          (button-2 (create-widget :push-button top-2
                    :label-string "Popup-2")))
     (add-wm-protocol-callback top-1 :wm-delete-window
                               #'xtk::quit-application shell)
     (add-wm-protocol-callback top-2 :wm-delete-window
                               #'xtk::quit-application shell)
     (add-callback button-1 :activate #'xtk::quit-application shell)
     (add-callback button-2 :activate #'xtk::quit-application shell)
     (popup top-1)
     (popup top-2)))
TWO-PRIMARY-WINDOWS

> (run-motif-application 'two-primary-windows)
```

`create-widget` (Function)

Syntax: `(defun create-widget (class parent &rest resources))`

Description: The function instantiates a managed widget of the given class and returns the widget ID for the newly created widget. The widget's name is equal to the symbol name of the widget's class with all upper case letters converted to lower case.

Note: CLM applications should use the `create-...` functions described in section 5.1.1 to create widgets.

To create a widget with a name different from the symbol name of the widget class, use the function `create-named-widget`. To create an unmanaged widget, use the function `create-unmanaged-widget`.

This function actually calls `create-named-widget` to instantiate the widget.

Parameters:

class	The widget class of the widget to be created.
parent	The widget ID of the parent widget.
resources	An arbitrary number of resource/value pairs.

`create-unmanaged-widget` (Function)

Syntax: `(defun create-unmanaged-widget (class parent &rest resources))`

Description: This function is identical to the `create-widget` function with the only difference that the newly created widget is unmanaged.

Note: CLM applications should use the `create-...` functions described in section 5.1.1 to create widgets.

Parameters:

class	The widget class of the widget to be created.
parent	The widget ID of the parent widget.
resources	An arbitrary number of resource/value pairs.

`create-named-widget` (Function)

Syntax: `(defun create-named-widget (class parent name managed resources))`

Description: The function instantiates a widget of the given class and returns the widget ID of the newly created widget.

Note: CLM applications should use the `create-...` functions described in section 5.1.1 to create widgets.

Parameters:

class	The widget class of the widget to be created.
parent	The widget ID of the parent widget.
name	The name of the widget to be created.
managed	Specifies whether the widget is managed.
resources	A list containing an arbitrary number of resource and value pairs.

`destroy-application` (Function)

Syntax: `(defun destroy-application (widget))`

Description: This function destroys the widget hierarchy pointed to by the given widget ID and causes the CLM server to terminate upon completion of the callback handler which contains the call to `destroy-application`.

Note: This function is automatically called when the predefined callback function `quit-application` is executed. Normally, there is no need for a CLM application to call this function.

This function must be called from inside a callback handler which in turn must be called from somewhere inside the call to `run-motif-application` which has instantiated the shell widget that is to be destroyed. It is therefore not possible for an application to destroy another application's widget hierarchy.

If a window manager other that `mwm` is used, the application is responsible for calling `destroy-application`. This is normally achieved by creating a *Quit* button and attaching the callback handler `quit-application` to it (see example below).

Parameters: *widget* Specifies the widget ID of the application shell which should be destroyed.

Example:

```
> (defun destroy-with-extra-button (&aux shell quit-button)
     "An application with an extra quit button"
     ;; Assume mwm is not running
     (setq shell (create-application-shell :allow-shell-resize t))
     (setq button (create-push-button shell "button"
      :label-string "Quit"))
     ;; quit-application is a predefined callback function (see text)
     (add-callback button :activate 'xtk::quit-application shell)
     (realize-widget shell))
destroy-with-extra-button

> (run-motif-application 'destroy-with-extra-button)
```

`destroy-widget` (Function)

Syntax: `(defun destroy-widget (widget))`

Description: This function is used to destroy a widget and all its children.

Parameters: *widget* Specifies the widget to be destroyed.

5.2 Dialogs

The X Toolkit Intrinsics and the Motif toolkit provide two types of dialogs:

- Dialogs which use a Motif `XmDialogShell` for their shell widget.
- Dialogs which use one of the X Toolkit Intrinsics shell classes.

5.2.1 Using `XmDialogShell` Widgets

This section describes functions which should be used to handle dialogs which use an `XmDialogShell` for their shell widget. Then the `XmDialogShell` must contain a widget of class `XmBulletinBoard` (or subclass), which must be created *unmanaged*. These dialogs are popped up and down by calling the functions `manage-popup-child` and `unmanage-popup-child` with the `XmBulletinBoard`'s widget ID as their first argument. CLM applications may also use the predefine callback functions `manage-popup` and `unmanage-popup` to pop dialogs up and down. These callback functions need the `XmBulletinBoard`'s widget ID as their client-data.

CLM applications should use the `create-....-dialog` functions described in tables 5.2 and 5.3 to create dialogs. These functions create an `XmDialogShell` with an unmanaged `XmBulletinBoard` (or subclass) child and return these widget ID's as multiple values (`XmBulletinBoard`'s widget ID first value !).

`manage-popup-child` (Function)

Syntax: `(defun manage-popup-child (widget))`

Description: This function must be used to pop up dialogs which are contructed using the Motif classes `XmDialogShell` and `XmBulletinBoard`. The `widget` argument must be the widget ID of the `XmBulletinBoard` widget. The function manages the `XmBulletinBoard` widget and handles all callbacks which result from popping up the dialog.

Parameters: *widget* Specifies the widget ID of the `XmBulletinBoard` (or subclass) widget of the dialog that is to be popped up.

`unmanage-popup-child` (Function)

Syntax: `(defun unmanage-popup-child (widget))`

Description: This function must be used to pop down dialogs which are contructed using the Motif classes `XmDialogShell` and `XmBulletinBoard`. The `widget` argument must be the widget ID of the `XmBulletinBoard` widget. The function unmanages the `XmBulletinBoard` widget and handles all callbacks which result from popping down the dialog.

Parameters: *widget* Specifies the widget ID of the `XmBulletinBoard` widget of the dialog that is to be popped down.

`manage-popup` (Function)

Syntax: `(defun manage-popup (widget client-data &rest call-data))`

Description: This function is a predefined callback handler which can be used to pop up dialogs which are contructed using the Motif classes `XmDialogShell` and `XmBulletinBoard`. The function actually calls `manage-popup-child` to pop up the dialog. The widget ID of the `XmBulletinBoard` widget must be passed in by the `client-data` argument.

Parameters:

widget	The ID of the widget which executes the callback handler.
client-data	The widget ID of the `XmBulletinBoard` widget that is to be managed.
call-data	Not used.

`unmanage-popup` (Function)

Syntax: `(defun unmanage-popup (widget client-data &rest call-data))`

Description: This function is a predefined callback handler which can be used to pop down dialogs which are contructed using the Motif classes `XmDialogShell` and `XmBulletinBoard`. The function actually calls `unmanage-popup-child` to pop down the dialog. The widget ID of the `XmBulletinBoard` widget must be passed in by the `client-data` argument.

Parameters:

widget	The ID of the widget which executes the callback handler.
client-data	The widget ID of the `XmBulletinBoard` widget that is to be unmanaged.
call-data	Not used.

5.2.2 Using the X Toolkit Shell Classes

The functions described in this section should only be used to create dialogs which use one of the shell classes defined by the X Toolkit Intrinsics. These classes are `:toplevel-shell`, `:transient-shell` and `:override-shell`. These dialogs must be popped up and down using the functions `popup` and `popdown`. CLM applications may also use the predefined callback functions `popup-none`, `popup-exclusive` and `popup-nonexclusive` to pop up these type of dialogs. These callback functions need the shell widget ID as their client-data.

`create-popup-shell` (Function)

Syntax: `(defun create-popup-shell (widget-name class parent &rest resources))`

Motif function: `XtCreatePopupShell()`

Description: This function is used to create a Pop-up shell widget. It returns the widget ID of the newly created shell widget. For further information on pop-up dialogs see [4], page 66.

Note: Applications should use the functions `create-toplevel-shell`, `create-transient-shell` and `create-override-shell` to create shell widgets.

Parameters:

widget-name	The textual name of the popup shell widget.
class	The widget class of the shell widget.
parent	The widget ID of the parent widget.
resources	An arbitrary number of resource/value pairs.

`popup` (Function)

Syntax: `(defun popup (widget &optional (grab-kind :grab-nonexclusive)))`

Motif function: `XtPopup()`

Description: This function is used to pop up a shell widget which was previously create by `create-popup-shell`, `create-toplevel-shell`, `create-transient-shell` and `create-override-shell` (see [4], page 67).

Parameters:

widget	Specifies the shell widget.
grab-kind	Specifies the grabkind. Possible values are `:grab-none`, `:grab-exclusive` and `:grab-nonexclusive`.

`popdown` (Function)

Syntax: `(defun popdown (widget))`

Motif function: `XtPopdown()`

Description: This function is used to pop down a shell widget which was previously popped up with the function `popup`. (see [4], page 69).

Parameters:

widget	Specifies the shell widget.

5.3 Menus

CLM applications should use the functions `create-popup-menu`, `create-pulldown-menu` and `create-option-menu` to create menus (see tables 5.2 and 5.3). Consult the Motif documentation on how to create menu systems and option menus.

Popup menus must be popped up and down using the functions described in section 5.2.1. Program 3 shows a small program which uses a popup menu. The event handler receives a description of the button press event on it's `call-data` argument. The popup menu is positioned by setting the x and y position of the menu pane.

```
(in-package 'demos)
(use-package 'xtk)

;; Pop up a menu when a button press event occurs in the label widget

(defun popup-menu ()
 (let* ((shell (create-application-shell))
        (label (create-widget :label shell :label-string "Popup"))
        ;; Use left button for selections
        (menu (create-popup-menu label "Menu" :which-button 1))
        (button-1 (create-push-button menu "Red"))
        (button-2 (create-push-button menu "Yellow"))
        (button-3 (create-push-button menu "Green"))
        (button-4 (create-push-button menu "Blue")))
   (add-event-handler label :button-press-mask #'do-popup menu)
   (realize-widget shell)))

(defun do-popup (widget client-data &rest call-data)
  (declare (ignore widget))
  (set-values client-data :x (fourth call-data) :y (fifth call-data))
  (manage-widgets client-data))

> (run-motif-application 'popup-menu)
```

Programm 3: Example for popup menus

5.4 Manipulating Widgets

The following functions are used to manipulate a widget's state. They map to their corresponding function in the X Toolkit Intrinsics and have no additional functionality.

`realize-widget` (Function)

Syntax: `(defun realize-widget (widget))`

Motif function: `XtRealizeWidget()`

Description: This function realizes a widget tree.

Parameters:	*widget*	Specifies the root widget ID of the subtree that should be realized.

`manage-widgets` (Function)

Syntax: `(defun manage-widgets (&rest widgets))`

Motif function: `XtManageChildren()`

Description: This function switches an arbitrary number of widgets into the managed state.

Note: This function must not be used to pop up dialogs. Applications should use `manage-popup-child` for this purpose.

Parameters:	*widgets*	Specifies the widgets that should be managed.

`unmanage-widgets` (Function)

Syntax: `(defun unmanage-widgets (&rest widgets))`

Motif function: `XtUnmanageChildren()`

Description: This function switches an arbitrary number of widgets into the unmanaged state.

Note: This function must not be used to pop down dialogs. Applications should use `unmanage-popup-child` for this purpose.

Parameters:	*widgets*	Specifies the widgets that should be unmanaged.

`map-widgets` (Function)

Syntax: `(defun map-widgets (&rest widgets))`

Motif function: `XtSetMappedWhenManaged()`

Description: This function maps an arbitrary number of widgets.

Parameters:	*widgets*	Specifies the widgets that should be mapped.

`unmap-widgets` (Function)

Syntax: `(defun unmap-widgets (&rest widgets))`

Motif function: `XtSetMappedWhenManaged()`

Description: This function unmaps an arbitrary number of widgets.

Parameters: *widgets* Specifies the widgets that should be unmapped.

---

set-insensitive (Function)

Syntax: `(defun set-insensitive (&rest widgets))`

Motif function: `XtSetSensitivity()`

Description: This function switches an arbitrary number of widgets into the insensitive state.

Parameters: *widgets* Specifies the IDs of the widgets that should be made insensitive.

---

set-sensitive (Function)

Syntax: `(defun set-sensitive (&rest widgets))`

Motif function: `XtSetSensitivity()`

Description: This function switches an arbitrary number of widgets into the sensitive state.

Parameters: *widgets* Specifies the IDs of the widgets that should be made sensitive.

---

get-parent (Function)

Syntax: `(defun get-parent (widget))`

Motif function: `XtParent()`

Description: This function returns the CLM widget ID of the given widget's parent. If the widget's parent wasn't created through a call to a CLM function, this functions assigns a new CLM widget ID to the widget's parent and returns this ID.

Parameters: *widget* Specifies the widget ID.

---

xt-translate-coordinates (Function)

Syntax: `(defun xt-translate-coordinates (widget x y))`

Description: This function takes a pair of coordinates relative to a widget and returns a pair of absolute coordinates as a list.

Parameters:	*widget*	Specifies the widget ID.
	x	Specifies the relative x-coordinate.
	y	Specifies the relative y-coordinate.

Chapter 6

Callback Functions

The functions described in this chapter are used to add and remove callback handlers. To add a callback handler to a callback list, use the `add-callback` function. To remove a callback handler from a callback list, use the `remove-callback` function. To remove all callbacks from a callback list, use the `remove-all-callbacks` function.

CLM function	OSF/Motif function
`add-callback`	`XtAddCallback()`
`remove-callback`	`XtRemoveCallback()`
`remove-all-callbacks`	`XtRemoveAllCallbacks()`

Table 6.1: Callback functions

Callback lists are identified by keywords. The keyword is derived from the Motif name by stripping the `XmN` prefix and the `Callback` suffix and by replacing all upper case letters with a hyphen followed by the corresponding lower case letter. Example: The Lisp name of the callback list `XmNactivateCallback` is `:activate`. Refer to the manual pages for the OSF/Motif widget classes for an overview of the supported callback lists.

The callback handler function can be any ordinary lisp function, a lambda expression or a CLOS method. In general, callback functions should be defined as follows:

```
(defun callback-function (widget client-data &rest call-data)
  ... )
```

The `widget` parameter contains the widget ID of the widget which executes the callback handler. The `client-data` parameter contains the client data that was passed to the `add-callback` function when the callback was added. The `client-data` argument may be any lisp object. The `call-data` argument contains data which depends on the widget's class and on the type of the callback list. The format of the `call-data` arguments for the supported widget classes is decribed in section 6.1.

If the format of the `call-data` argument is known in advance, then the callback functions may be declared with an argument list which matches exactly the number of elements in the `call-data` argument. For example, the `:ok` callback of the `XmFileSelectionBox` provides it's callback functions with four call data arguments: the full file name, the current directory mask, the current base directory and the current file search pattern. Therefore, a callback function which handles an `:ok` callback may also be declared as

```
(defun ok-callback-function (widget client-data
                             file-name dir-mask base-dir file-pattern)
  ... )
```

`add-callback` (Function)

Syntax: `(defun add-callback (widget callback-name function client-data))`

Description: This function adds a callback handler to a callback list of a widget.

Parameters:

widget	Specifies the ID of the widget where a callback should be added.
callback-name	Specifies the name of the callback list.
function	Specifies the callback handler function.
client-data	Specifies the client data which should be passed to the callback handler function. The client data may be any lisp object.

`remove-callback` (Function)

Syntax: `(defun remove-callback (widget callback-name function client-data))`

Description: This function removes a callback handler from a widget's callback list. The client data passed to `remove-callback` must be `equal` to the client data that was passed to `add-callback` when the callback was added to the callback list.

Parameters:

widget	Specifies the ID of the widget where a callback should be removed.
callback-name	Specifies the name of the callback list.
function	Specifies the callback handler function.
client-data	Specifies the client data.

`remove-all-callbacks` (Function)

Syntax: `(defun remove-all-callbacks (widget callback-name))`

Description: This function removes all callback handlers from a widget's callback list.

Parameters:

widget	Specifies the ID of the widget where the callbacks should be removed.
callback-name	Specifies the name of the callback list from which to remove all callback handlers.

6.1 Contents of the `call-data` argument

If not stated otherwise, Callback functions have no `call-data` argument. Callback functions which declare their `call-data` argument as a `&rest` argument will then have an empty list on this argument. Callback functions which receive no `call-data` argument may also be defined as

```
(defun callback-function-with-no-call-data (widget client-data)
  ... )
```

6.1.1 The Class `:command` (`XmCommand`)

All callbacks have two call data arguments which contain the command string and the length of the command string. Callback functions may be defined as

```
(defun command-callback-function (widget client-data command length)
  ... )
```

6.1.2 The Class `:drawing-area` (`XmDrawingArea`)

The `:resize` callback delivers a single call data argument which contains the X window ID of the window associated with the `XmDrawingArea` widget. Callback functions may be defined as

```
(defun drawing-area-callback-function (widget client-data x-window-id)
  ... )
```

The `:expose` callback delivers six call data arguments: The X window ID of the window associated with the `XmDrawingArea` widget, and the `x`, `y`, `width`, `height` and `count` members from the expose event structure. Callback functions may be defined as

```
(defun drawing-area-expose-callback (widget client-data x-window-id
                                      x y width height count)
  ... )
```

The `:input` callback delivers a variable number of call data arguments which depend on the actual type of the input event. The first call data argument is the X window ID of the window associated with the `XmDrawingArea` widget, the second argument is a keyword which indicates the type of input event:

- For button events, this keyword is either `:button-press` or `:button-release`. There are seven additional call data arguments for button events: The `x`, `y`, `x_root`, `y_root`, `state`, `button` and `time` members from the `xbutton` event structure.
- For key events, the keyword is either `:key-press` or `:key-release`. There are seven additional call data arguments for events: The `x`, `y`, `x_root`, `y_root`, `state`, `key` members from the `xkey` event structure and a translation of the `key` member to it's string representation.

Applications should define callback functions which handle `:input` callbacks as

```
(defun drawing-area-input-callback (widget client-data x-window-id
                                     type x y x-root y-root state
                                     &rest additional-call-data)
  ... )
```

The callback handler should first check for the actual input event type by examining the `type` argument and then access the elements in the `additional-call-data` argument.

The X window ID may be converted to a CLX window ID with the function `make-clx-window`.

6.1.3 The Class `:drawn-button` (`XmDrawnButton`)

All callbacks except the `:expose` callback deliver a single call data argument which contains the X window ID of the window associated with the `XmDrawnButton` widget. Callback functions may be defined as

```
(defun drawn-button-callback-function (widget client-data x-window-id)
  ... )
```

The `:expose` callback delivers six call data arguments: The X window ID of the window associated with the `XmDrawnButton` widget, and the `x`, `y`, `width`, `height` and `count` members from the expose event structure. Callback functions may be defined as

```
(defun drawn-button-expose-callback (widget client-data x-window-id
                                      x y width height count)
  ... )
```

The X window ID may be converted to a CLX window ID with the function `make-clx-window`.

6.1.4 The Class `:file-selection-box` (`XmFileSelectionBox`)

All callbacks except the `:ok` and `:apply` callback have no `call-data` argument. The `:ok` and the `:apply` callback have four call data arguments: the full file name, the current directory mask, the current base directory and the current file search pattern. Callback functions may be declared as

```
(defun ok-callback-function (widget client-data
                             file-name dir-mask base-dir file-pattern)
  ... )
```

6.1.5 The Class `:list` (`XmList`)

The contents of the `XmList` widget's callbacks is shown in table 6.2.

The `item` argument contains the selected item. The `position` argument contains the item's position in the list of items. For callbacks of type `:extended-selection`, the `type` argument describes the selection type. It may be one of `:initial`, `:addition` and `:modification`. These values correspond to the Motif values `XmINITIAL`, `XmADDITION` and `XmMODIFICATION`. The argument `n` is the number of selected items. The arguments

Callback name	call-data
`:browse-select`	`item position`
`:default-action`	`item position`
`:single-select`	`item position`
`:extended-selection`	`type n item`$_1$ `...` `item`$_n$
`:multiple-selection`	`n item`$_1$ `...` `item`$_n$

Table 6.2: Callbacks of the `XmList` widget class

`item`$_1$ `...` `item`$_n$ are the selected items. For example, callback functions which handle an `:extended-selection` callback may be declared as

```
(defun extended-selection (widget client-data type n &rest selected-items)
  ... )
```

6.1.6 The Class `:scale` (`XmScale`)

The call data argument of all callbacks is a single argument which contains the new slider position. Callback functions may be declared as

```
(defun scale-callback-function (widget client-data new-slider-position)
  ... )
```

6.1.7 The Class `:scroll-bar` (`XmScrollBar`)

All callbacks except `:to-bottom` and `:to-top` receive a single argument which is the new slider value. The `:to-bottom` and `:to-top` callbacks receive two arguments. The first is the new slider position, the second is the relative pixel coordinate where the mouse button was pressed. (see [5], page 444). Callback functions which handle a `:drag` callback may be defined as

```
(defun drag-callback-function (widget client-data new-slider-value)
  ... )
```

6.1.8 The Class `:selection-box` (`XmSelectionBox`)

The `:ok` callback has a single call data argument which contains the selected string item. Callback functions may be defined as

```
(defun selection-callback-function (widget client-data selected-item)
  ... )
```

6.1.9 The Class `:text` (`XmText`) and `:text-field` (`XmTextField`)

The contents of the call data arguments for the Motif widget classes `XmText` and `XmTextField` is shown in table 6.3,

The `:value-changed` call data for multi-line text widgets is always `nil`. The text string may be retrieved with the `get-values` function. For single-line `XmText` widgets

Callback name	`call-data`
`:motion-verify`	`current-insert new-insert`
`:losing-focus`	`current-insert new-insert`
`:modify-verify`	`current-insert new-insert start-pos end-pos text length`
`:value-changed`	`value` or `nil` (see text)
`:activate`	`value` or `nil`

Table 6.3: Callback-Funktionen der Klasse `text`

and for `XmTextField` widgets, the call data argument of the `:value-changed` callback contains the widget's text string.

The return value of callback functions which handle `:modify-verify` and `:motion-verify` callbacks is used to determine whether a text modification or a cursor movement is allowed or denied. A return value of `nil` denies, a non-nil return value allows a modification or a cursor movement.

6.1.10 The Class `:toggle-button` (`XmToggleButton`)

The call data argument of all callbacks is a single argument which contains the state of the toggle button (`t` or `nil`). Callback functions may be defined as

```
(defun toggle-callback-function (widget client-data state)
  ... )
```

6.2 An Example on Using Callbacks

Program 4 demonstrates the use of callback functions. The widget tree consists of a shell widget, a RowColumn widget, two ToggleButtons and two Text widgets. The state of the two ToggleButtons is displayed in the text widgets. Callbacks are added to the ToggleButtons which update the contents of the text widgets when the state of the button changes. The first ToggleButton uses the callback handler function `show-state` to update the text in the Text widget. The second button uses a lambda expression to update the text. Both callback handlers receive the ID of the text widget which they should use for displaying their state by their `client-data` parameter.

The call data the ToggleButton passes to the callback handler functions is either `t` or `nil` and describes the state of the button. The callback handler of the first button declares the call data as a `&rest` parameter, the second callback handler declares the call data as an ordinary parameter. These second alternative is possible because the format of the call data is known in advance.

```
> (defun callback-demo (&aux shell row-column toggle-1 text-1 toggle-2 text-2)
   (setf shell (create-application-shell)
         row-column (create-row-column shell "row")
         toggle-1 (create-toggle-button row-column "toggle-1" :set nil)
         text-1 (create-text-field row-column "text-1"
                                   :value "Off" :editable nil)
         toggle-2 (create-toggle-button row-column "toggle-2" :set t)
         text-2 (create-text-field row-column "text-2"
                                   :value "On" :editable nil))
   (add-callback toggle-1 :value-changed #'show-state text-1)
   (add-callback toggle-2 :value-changed
                 #'(lambda (widget client-data &rest call-data)
                     (set-values client-data
                                 :value (if (first call-data) "On" "Off")))
                 text-2)
   (realize-widget shell))

CALLBACK-DEMO
> (defun show-state (widget client-data &rest call-data)
    (set-values client-data :value (if (first call-data) "On" "Off")))

SHOW-STATE
> (run-motif-application 'callback-demo)

NIL
```

Programm 4: The program `Callback Demo`

Chapter 7

Protocol Callbacks

The functions described in this chapter are used to add and remove protocol callbacks from shell widgets. The functions provide the functionality of the Motif functions `XmAddProtocolCallback()`, `XmAddWMProtocolCallback()`, `XmRemoveProtocolCallback()` and `XmRemoveWMProtocolCallback()`. For more information on protocol callbacks see [5], pages 99-100 and 102-103, and [8], and [2], pages 43–46.

The X atoms used to identify the protocol and property must be given as symbols. They are automatically converted to X atoms by the CLM server.

CLM function	OSF/Motif function
`add-protocol-callback`	`XmAddProtocolCallback()`
`add-wm-protocol-callback`	`XmAddWMProtocolCallback()`
`remove-protocol-callback`	`XmRemoveProtocolCallback()`
`remove-wm-protocol-callback`	`XmRemoveWMProtocolCallback()`

Table 7.1: Callback functions

`add-protocol-callback` (Function)

Syntax:

```
(defun add-protocol-callback
       (widget property protocol function client-data))
```

Description: This function adds a protocol callback to a shell widget.

Parameters:

widget Specifies the ID of the widget where to add a protocol callback.

property Specifies the name of the property as a keyword symbol.

protocol Specifies the name of the protocol as a keyword symbol.

function Specifies the protocol callback handler function.

client-data Specifies the client data which should be passed to the callback handler function.

Example: The example shows the implementation of the function `create-application`. The function adds a protocol callback to the newly created shell widget. The protocol callback is executed when the user selects the *Close* command of the `mwm` window manager. The keywords `:wm-protocols` and `:wm-delete-window` correspond to the X property name `"WM_PROTOCOLS"` and the protocol `"WM_DELETE_WINDOW"`.

```
(defun create-application-shell (&rest resources &aux res wid)
  (setq res (append resources (list :delete-response :do-nothing)))
  (setq wid (car (convenience-function 0 1 0 "" res)))
  (when (is-mwm-running wid)
    (add-wm-protocol-callback wid :wm-delete-window
      #'quit-application wid))
  wid)
```

`add-wm-protocol-callback` (Function)

Syntax:

```
(defun add-wm-protocol-callback
       (widget protocol function client-data))
```

Description: This function adds a window manager protocol callback to a shell widget. The functions actually calls `add-protocol-callback` with the property argument set to `:wm-protocols`.

Parameters:

widget Specifies the ID of the widget where to add a protocol callback.

protocol Specifies the name of the protocol as a keyword symbol.

function Specifies the protocol callback handler function.

client-data Specifies the client data which should be passed to the callback handler function.

`remove-protocol-callback` (Function)

Syntax:

```
(defun remove-protocol-callback
       (widget property protocol function client-data))
```

Description: This function removes a protocol callback from a shell widget.

Parameters:

widget Specifies the ID of the widget where to remove a protocol callback.

property Specifies the name of the property as a keyword symbol.

protocol	Specifies the name of the protocol as a keyword symbol.
function	Specifies the protocol callback handler function.
client-data	Specifies the client data which should be passed to the callback handler function.

`remove-wm-protocol-callback` (Function)

Syntax:

```
(defun remove-wm-protocol-callback
       (widget protocol function client-data))
```

Description: This function removes a window manager protocol callback from a shell widget. The functions actually calls `remove-protocol-callback` with the property argument set to `:wm-protocols`.

Parameters:

widget	Specifies the ID of the widget where to remove a protocol callback.
protocol	Specifies the name of the protocol as a keyword.
function	Specifies the protocol callback handler symbol.
client-data	Specifies the client data which should be passed to the callback handler function.

Chapter 8

Event Handlers

The functions described in this chapter are used to add and remove event handlers for widgets. Refer to the X Toolkit documentation for more information about event handlers.

In general, an event handler function should be defined as follows:

```
(defun event-handler (widget client-data event-type &rest event)
   ... )
```

The `widget` argument contains the ID of the widget which executes the event handler. The `client-data` argument contains the client data that was passed to `add-event-handler` when the event handler was installed. The `event-type` argument specifies the X event type. The `event` argument contains data from the X event structure. The format of this data is decribed in table 8.1.

X Event type (see [2], page 145)	Format of the `&rest event` argument (see [2], pages 146–182)
KeyPress, KeyRelease	`(x y x-root y-root state keycode keysym)`
ButtonPress, ButtonRelease	`(x y x-root y-root state button time)`
Expose	(n-events x_1 y_1 width$_1$ height$_1$ ... x_n y_n width$_n$ height$_n$)
MotionNotify	(n-events x_1 y_1 x-root$_1$ y-root$_1$ state$_1$ ... ... x_n y_n x-root$_n$ y-root$_n$ state$_n$)
EnterNotify, LeaveNotify	`(x y x-root y-root mode detail focus state)`
FocusIn, FocusOut	`(mode detail)`
ConfigureNotify	`(x y width height)`
VisibilityNotify	`(state)`
PropertyNotify	`(atom-id state time)`
SelectionClear	`(selection-atom time)`
SelectionRequest	`(requestor slection target property time)`
SelectionNotify	`(selection target propery time)`
ClientMessage	`(message-type <5, 10, or 20 numbers>)`

Table 8.1: Format of the event data for event handlers

For efficiency, the CLM server buffers `Expose` events for each widget and calls the installed event handler only when the `count` field in the expose event is zero. The `event` argument of the event handler then contains the number of buffered expose events and the `x`, `y`, `width` and `height` for each expose event. Expose events are only buffered if the `compress_exposure` field in the widget's class record is `False`. CLM applications may code their expose event handler as follows:

```
(defun expose-event-handler (widget client-data event-type &rest event)
  (setq regions (cdr events))
  (loop for n-events from 1 to (first event) do
    (update-region widget client-data
                   (first regions) (second regions)
                   (third regions) (fourth regions))
    (setq regions (cddddr regions))))
```

The CLM server also calls event handlers for `Motion` events only once if it receives more than one `Motion` event at a time. The `event` argument of the event handler then contains the number of `Motion` events and the `x`, `y`, `x_root`, `y_root` and `state` fields for each `Motion` event. CLM applications may code their `Motion` event handler as follows:

```
(defun motion-event-handler (widget client-data event-type &rest event)
  (setq motions (cdr events))
  (loop for n-events from 1 to (first event) do
    (track-mouse widget client-data
                 (first motions) (second motions)
                 (third motions) (fourth motions) (fifth motions))
    (setq motions (nthcdr 5 motions))))
```

Event types are specified as keywords. The keyword can be derived from the X event name (see [2], pages 144-146) by inserting a hyphen before each upper case letter and by converting the upper case letters to lower case. For example, to install an event handler which reacts to key events, the event mask `'(:key-press-mask :key-release-mask)` must be used.

`add-event-handler` (Function)

Syntax:

```
(defun add-event-handler (widget event-mask function client-data
                          &key (non-maskable nil)))
```

Description: This function installs an event handler for a widget. An event handler may be installed for a single event type or for multiple event types. In the first case, the event name must be given as an atom, in the second case, a list of event names must be given.

Parameters:

widget	Specifies the widget ID.
event-mask	Specifies the event name or a list of event names.
function	Specifies the event handler function.

client-data	Specifies the client data for the event handler function.
non-maskable	Specifies whether the event handler should also be installed for the non-maskable event types.

`remove-event-handler` (Function)

Syntax:

```
(defun add-event-handler (widget event-mask function client-data
                          &key (non-maskable nil)))
```

Description: This function removes an event handler from a widget. An event handler may be removed for a single event type or for multiple event types. In the first case, the event name must be given as an atom, in the second case, a list of event names must be given.

Parameters:

widget	Specifies the widget ID.
event-mask	Specifies the event name or a list of event names.
function	Specifies the event handler function.
client-data	Specifies the client data for the event handler function.
non-maskable	Specifies whether the event handler should also be removed for the non-maskable event types.

Chapter 9

Resources

The functions described in this chapter are used to set and retrieve the resources of widgets. These functions map to the Motif functions `XtSetValues()` and `XtGetValues()`.

When setting a widget's resource either at widget creation time or dynamically using `set-values`, the resources are specified as a sequence of arguments containing the resource names and resource values. For example:

```
(set-values widget :label-string "Good morning" :background "red")

(create-widget parent :text :value "Hello" :editable nil)
```

In the example above, the keywords `:label-string`, `:background`, `:value` and `:editable` denote the Motif resources `XmNlabelString`, `XmNbackground`, `XmNvalue` and `XmNeditable`, respectively. Their values are `"Good morning"`, `"red"`, `"Hello"` and `nil`, respectively.

Table 9.1 shows how Motif resources are specified in Lisp. The left column contains the "C" resource type. The right column contains their possible Lisp values. Table 9.2 lists the possible values for enumerative resources which have the "C" type `unsigned char`. The left column contains the Motif resource class and the right column contains the possible values for this class.

Refer to the Motif Programmer's Guide for which resources should be best specified in the program and which resources should be best specified in the application's default file.

`set-values` (Function)

Syntax:	`(defun set-values (widget &rest resources))`	
Description:	This function is used to set a widget's resources.	
Parameters:	*widget*	Specifies the widget ID.
	resources	Specifies an arbitrary number of name/value pairs.

Motif type	Lisp representation
Dimension, Integer Short	Fixnum
String, XmString	String
Widget	The widget ID
FontList	String name of the font
Boolean, Bool	`t nil`
Atom	Atom name as string
Pixel	String or rgb-value (e.g. `"red"` or `"#222333444"`)
Pixmap	Path name of pixmap file
Translations	String
Char, UChar	String of length 1

Table 9.1: Resource types and values

`get-values` (Function)

Syntax: `(defun get-values (widget &rest resource-names))`

Description: This function is used to retrieve an arbitrary number of widget resources. The resource names are given with the `resource-names` argument. The function returns a list which contains the resource values. The order of the resource values in the returned list corresponds to the order of the resource names in the argument list.

Parameters:

widget	Specifies the widget ID.
resources	Specifies the resource names.

Motif resource class	Lisp resource values
XmCAlignment	:center :end :beginning
XmCArrowDirection	:arrow-down :arrow-left :arrow-right :arrow-up
XmCAttachment	:form :position :widget :self :none :opposite-form :opposite-widget
XmCCommandWindowLocation	:above-workspace :below-workspace
XmCDefaultButtonType	:cancel :ok :help
XmCDeleteResponse	:destroy :unmap :do-nothing
XmCDialogStyle	:system-modal :application-modal :modeless :work-area
XmCDialogType	:file-selection :prompt :selection :work-area :error :message :question :warning :information
XmCEditMode	:multi-line-edit :single-line-edit
XmCFileTypeMask	:regular :directory :any-type
XmCIndicatorType	:one-of-many :n-of-many
XmCInitialState	:iconic :normal
XmCKeyboardFocusPolicy	:pointer :explicit
XmCLabelType	:string :pixmap
XmCListSizePolicy	:constant :variable :resize-if-possible
XmCMultiClick	:keep :discard
XmCNavigationType	:none :sticky-tab-group :tab-group :exclusive-tab-group
XmCOrientation	:horizontal :vertical
XmCPacking	:pack-tight :pack-none :pack-column
XmCProcessingDirection	:max-on-top :max-on-left :max-on-right :max-on-bottom
XmCResizePolicy	:any :grow :none
XmCRowColumnType	:work-area :menu-bar :menu-pulldown :menu-popup :menu-option
XmCScrollBarDisplayPolicy	:as-needed :static
XmCScrollBarPlacement	:top-left :top-right :bottom-left :bottom-right
XmCScrollingPolicy	:automatic :application-defined
XmCSelectionPolicy	:single :multiple :extended :browse
XmCSeparatorType	:single-line :double-line :double-dashed-line :single-dashed-line :no-line :shadow-etched-in :shadow-etched-out
XmCShadowType	:in :out :etched-in :etched-out
XmCStringDirection	:l-to-r :r-to-l
XmCUnitType	:100th-millimeters :1000th-inches :100th-points :100th-font-units :pixels
XmCVisualPolicy	:constant :variable

Table 9.2: Enumerative resource classes and values

Chapter 10

Translations

The functions described in this chapter are used to manipulate a widget's translation tables. The functions take a string argument which describes the translation table. The syntax is the same as for the X Toolkit Intrinsics. The strings are automatically converted to translation tables suitable for `XmAugmentTranslations()` and `XmOverrideTranslations()`.

CLM function	OSF/Motif function
`augment-translations`	`XtParseTranslationTable()` `XtAugmentTranslations()`
`override-translations`	`XtParseTranslationTable()` `XtOverrideTranslations()`
`push-translations`	*none*
`pop-translations`	*none*

Table 10.1: Translation table functions

`augment-translations` (Function)

Syntax: `(defun augment-translations (widget translation-table))`

Description: This function is used to augment a widget's translations.

Parameters:

widget	Specifies the widget ID.
translation-table	Specifies the translation table.

`override-translations` (Function)

Syntax: `(defun override-translations (widget translation-table))`

Description: This function is used to override a widget's translations.

Parameters:	*widget*	Specifies the widget ID.
	translation-table	Specifies the translation table.

push-translations (Function)

Syntax: `(defun push-translations (widget))`

Description: This function replaces a widget's translations with the empty translation table until the old translation table is restored with `pop-translations`. This disables the widget from dispatching any events except expose events.

Parameters: *widget* Specifies the widget ID.

pop-translations (Function)

Syntax: `(defun pop-translations (widget))`

Description: This function restores the original translation table of a widget.

Parameters: *widget* Specifies the widget ID.

Chapter 11

Tab Groups

The functions in this chapter are used to add and remove tab groups and to simulate keyboard traversal actions. Refer to the Motif documentation for details.

CLM function	OSF/Motif function
`add-tab-group`	`XmAddTabGroup()`
`remove-tab-group`	`XmRemoveTabGroup()`
`process-traversal`	`XmProcessTraversal()`

Table 11.1: Tab group functions

`add-tab-group` (Function)

Syntax: `(defun add-tab-group (widget))`

Description: This function adds a tab group to a shell widget.

Parameters: *widget* Specifies the widget ID of the tab group

---

`remove-tab-group` (Function)

Syntax: `(defun remove-tab-group (widget))`

Description: This function removes a tab group from a shell widget.

Parameters: *widget* Specifies the widget ID tof the tab group.

`process-traversal` (Function)

Syntax: `(defun process-traversal (widget direction))`

Motif function: `XmProcessTraversal()`

Description:	This function moves the keyboard focus to another widget. It returns `t` if the traversal action could be performed and `nil` otherwise.	
Parameters:	*widget*	Specifies the widget ID of the focus widget.
	direction	Specifies the direction of traversal. Possible values are `:current`, `:down`, `:home`, `:left`, `:next`, `:next-tab-group`, `:prev`, `:prev-tab-group`, `:right` and `:up`.

Chapter 12

Timers

The functions described in this chapter are used to create and manipulate timers. These functions provide a high level interface to the functions `XtAddTimeout()` and `XtRemoveTimeout()` which are defined by the X Toolkit Intrinsics. Examples on the usage of timers are discussed in the examples in sections 4.7.1 and 4.7.2.

CLM function	Description
`create-timer`	Create a timer and return it's ID
`destroy-timer`	Destroy a timer
`change-timer`	Change a timer's interval
`start-timer`	Start a timer
`stop-timer`	Stop a timer

Table 12.1: Timer functions

Timers are created using the function `create-timer`. This function returns a unique timer ID which may be used for further operations on the timer. When a timer is created, the programmer must specify an initial timer interval (in milliseconds), a handler function which is called each time the time interval elapses and a client data argument which is passed to the timer function. The `create-timer` function also provides several keyword arguments which can be used to modify the general behavior of a timer.

Timer handler functions must be declared as follows:

```
    (defun timer-handler (timer-id client-data)
... )
```

The `timer-id` argument is the ID of the timer which executes the timer handler function. The `client-data` argument contains the client data which was passed to the function `create-timer` when the timer was created.

The timer handler function is called each timer the timer interval expires. By default, the next timer interval starts immediately after the execution of the timer handler. This behavior can be changed with the `:active-in-handler` argument of the function `create-timer`. If this argument's value is non-nil, then the next time interval starts immediately after the previous interval has elapsed.

A timer starts running immediately after it has been created. To create a timer which is initially disabled, the `:enabled` argument of the function `create-timer` must be given a value of `nil`. The timer is started with the function `start-timer`. A timer is stopped with the function `stop-timer`. A timer is destroyed with the function `destroy-timer`. It is always save to start, stop and destroy a timer during the execution of it's timer callback function.

`create-timer` (Function)

Syntax:

```
(defun create-timer (interval function client-data
     &key (enabled t)
  (active-in-handler nil)
  (active-in-recursive-main-loop nil)
  (once nil)))
```

Description: This function creates a timer and returns it's timer ID.

Parameters:

interval	Specifies the timer interval in milliseconds
function	Specifies the timer handler function.
client-data	Specifies the client data which is is passed to the timer function each time it is executed.
enabled	Specifies whether the timer starts immediately after it has been created. A non-nil value creates a disabled timer, which must be started using the function `start-timer`.
active-in-handler	Specifies the beginning of the next timer interval. A non-nil value starts the next time interval immediately after the previous interval has elapsed. A value of `nil` starts the next time interval immediately after the execution of the timer handler function has finished.
active-in-recursive-main-loop	Specifies whether the timer handler should be called when CLM is executing a recursive application main loop. An implicit recursive main loop is executed during the execution of the functions `alert-dialog` and `wait-for-input`.
once	A non-nil value causes the timer to be destroyed automatically immediately after the first execution of the timer handler function.

`destroy-timer` (Function)

Syntax:

```
(defun destroy-timer (timer-id))
```

Description: This function destroys a timer.

Parameters:	*timer-id*	Specifies the timer.

`change-timer` (Function)

Syntax: `(defun change-timer (timer-id new-interval))`

Description: This function changes the time interval of a timer. The function doesn't affect the current time interval. The new time interval is used after the current interval has expired.

Parameters:	*timer-id*	Specifies the timer.
	new-interval	Specifies the new timer interval in milliseconds.

`start-timer` (Function)

Syntax: `(defun start-timer (timer-id &key (new-interval -1)))`

Description: This function starts a disabled timer. A value of -1 doesn't affect the current time interval.

Parameters:	*timer-id*	Specifies the timer.
	new-interval	Specifies an optional new timer interval in milliseconds.

`stop-timer` (Function)

Syntax: `(defun stop-timer (timer-id))`

Description: This function disables a running timer.

Parameters:	*timer-id*	Specifies the timer.

Chapter 13

Cursors

The functions described in this chapter are use to create cursors and to change a widget's cursor.

CLM function	OSF/Motif function
`create-font-cursor`	`XCreateFontCursor()`
`create-pixmap-cursor`	`XCreatePixmapCursor()`
`define-cursor`	`XDefineCursor()`
`change-grab-cursor`	`XChangeActivePointerGrab()`

Table 13.1: Cursor functions

To use a certain cursor shape for a widget, a cursor must first be created and then must be defined for the widget. CLM supports two cursor types:

- *Font cursors* are cursors from the X cursor font. The cursor shape is specified using a keyword symbol. A list of possible font cursors is returned by the function `xtk::all-cursors`. Font cursors are created with the function `create-font-cursor`.
- *Pixmap cursors* are constructed from two bitmaps of equal size. A description of the bitmaps must be stored in a file which can be read by the `XReadBitmapFile()` function from the X library (see [2]).

The functions which create cursors return an X cursor ID. This cursor ID must be used for defining a new cursor shape for a widget. The function `define-cursor` can be used to permanently define a new cursor shape for a widget. The function `change-grab-cursor` can be used to change a widget's cursor during an active pointer grab.

`create-font-cursor` (Function)

Syntax: `(defun create-font-cursor (shape))`

Description: This function creates a font cursor from the X cursor font and returns an X cursor ID.

Parameters:	*shape*	Specifies the font shape as a keyword. A list of all possible shapes is returned by the function `xtk::all-cursors`.

`create-pixmap-cursor` (Function)

Syntax:
```
(defun create-pixmap-cursor (source mask &key (foreground "black")
      (background "white")))
```

Motif function: `XCreatePixmapCursor()`

Description: This function creates a bitmap cursor from two bitmap files and returns an X cursor ID. The bitmaps must be stored in the X bitmap format. Bitmaps may be created by using the `bitmap` utility from the X distribution. The two bitmaps must have an equal size and the `source` bitmap must have a *hot spot* defined.

Parameters:	*source*	Specifies the pathname of the X bitmap file which contains the source bitmap.
	mask	Specifies the pathname of the X bitmap file which contains the mask bitmap.
	foreground	Specifies the cursors foreground color.
	background	Specifies the cursors background color.

`define-cursor` (Function)

Syntax:
```
(defun define-cursor (cursor-id &rest widgets))
```

Motif function: `XDefineCursor()`

Description: This function is used to define a new cursor shape for an arbitrary number of widgets. If the symbol `:restore` is used instead of a cursor shape, then the widget's old cursor shape is restored.

Parameters:	*cursor-id*	Specifies the cursor-id or `:restore`.
	mask	Specifies the widgets for which a new cursor should be defined.

`change-grab-cursor` (Function)

Syntax:
```
(defun change-grab-cursor (cursor-id &key (add-mask nil) (remove-mask nil)))
```

Motif function: `XChangeActivePointerGrab()`

Description:	This function is used to define a cursor shape during an active pointer grab. The two keyword arguments may be set to modify the X event mask used during the pointer grab. The two event masks have the same format as the event mask used for installing event handlers. The event mask are first converted to X event masks. The event mask used during the pointer grab is computed as	

```
    mask = (ButtonReleaseMask | ButtonMotionMask | add_mask)
   & ~ remove_mask
```

Parameters:	*cursor-id*	Specifies the cursor-id.
	add-mask	Specifies the events which should be reported during an active pointer grab.
	remove-mask	Specifies the events which should *not* be reported during an active pointer grab.

Chapter 14

Query functions

The functions described in this chapter are used to alert dialogs. These functions have no corresponding Motif functions.

`alert-dialog` (Function)

Syntax:	`(defun alert-dialog (widget))`	
Description:	This function pops up a dialog and blocks until the dialog is popped down. The dialog must be popped down using the `unmanage-popup` function. The dialog must be created using a `XmDialogShell` widget with and `XmBulletinBoard` child. The `widget` argument of `alert-dialog` must be the widget ID of the `XmBulletinBoard` widget.	
Note:	The `XmBulletinBoard` widget's `:auto-unmanage` resource must be set to `nil` at widget creation time. The `:unmap` callback list of the `XmBulletinBoard` widget must not be modified.	
Parameters:	*widget*	Specifies the widget ID of the `XmBulletinBoard` widget.

`wait-for-input` (Function)

Syntax:	`(defun wait-for-input (widget))`	
Description:	This function waits for the user to type in some text into a `XmText` or `XmTextField` widget and blocks until the user presses a key which causes the text widget's `:activate` callback to be executed. This callback is normally executed when the *Activate* key is pressed.	
Note:	The `XmText` widget's `:activate` callback list must not be modified. The widget's `:edit-mode` resource is forced to `:single-line-edit`.	
Parameters:	*widget*	Specifies the widget ID of the text widget.

Chapter 15

Interface to Widget Classes

The functions described in this chapter correspond to Motif functions which are exported by the Motif widget classes. Refer to the widget's manual pages for more information about these functions.

15.1 The `XmText` Widget Class

The functions described in this section provide an interface to the functions exported by the **`XmText`** widget class. The functions **`text-insert`** and **`text-append`** are used for inserting and appending text to a multi-line text widget.

`text-insert` (Function)

Syntax:

```
(defun text-insert (widget position text &key (move-cursor t)
                                              (scroll-window nil)))
```

Description: This functions inserts a text string in a text widget.

Parameters:

widget	Specifies the widget ID.
position	Specifies the position where to insert the text.
text	Specifies the text to be inserted.
move-cursor	Specifies whether the cursor should be positioned after the last inserted character.
scroll-window	Specifies whether the text widget should be scrolled to the position where the new text was inserted.

`text-append` (Function)

Syntax:

```
(defun text-append (widget text &key (move-cursor t)
                                     (scroll-window nil)))
```

Description: This function appends a string to a text widget.

Parameters:

widget	Specifies the text widget.

CLM function	OSF/Motif function
`text-insert`	`XmTextInsert`
`text-append`	`XmTextAppend`
`text-get-selection`	`XmTextGetSelection`
`text-clear-selection`	`XmTextClearSelection`
`text-cut`	`XmTextCut`
`text-copy`	`XmTextCopy`
`text-get-baseline`	`XmTextGetBaseline`
`text-get-selection-position`	`XmTextGetSelectionPosition`
`text-paste`	`XmTextpaste`
`text-pos-to-xy`	`XmTextPosToXY`
`text-remove`	`XmTextRemove`
`text-scroll`	`XmTextScroll`
`text-set-add-mode`	`XmTextSetAddMode`
`text-show-position`	`XmTextShowPosition`
`text-xy-to-pos`	`XmTextXYToPos`
`text-get-last-pos`	`XmTextGetLastPos`
`text-set-selection`	`XmTextSetSelection`
`text-replace`	`XmTextReplace`
`text-get-insertion-position`	`XmTextGetInsertionPosition`
`text-set-highlight`	`XmTextSetHighlight`
`text-get-substring`	none
`text-search`	none

Table 15.1: Functions for the `XmText` class

text — Specifies the text to be inserted.

move-cursor — Specifies whether the cursor should be positioned after the last inserted character.

scroll-window — Specifies whether the text widget should be scrolled to the position where the new text was inserted.

There are a number of other Motif text functions available. They are listed in Table 15.1. See the Motif documentation and the CLM source code (file `text.lisp`) for details.

15.2 The `XmList`, `XmCommand` and `XmSelectionBox` Widget Classes

The functions described in this section provide an interface to the exported functions of the `XmList` widget class. The functions `set-items`, `get-items` and `get-selected-items` may also be used with widgets of the classes `XmCommand` and `XmSelectionBox`.

`set-items` (Function)

Syntax: `(defun set-items (widget &rest items))`

Description: This function replaces the items of a widget by the items given in the `items` parameter. This function is applicable to widget of the classes `XmList`, `XmCommand` and `XmSelectionBox`.

Parameters:
widget Specifies the widget ID.
items Specifies the new items.

`get-items` (Function)

Syntax: `(defun get-items (widget))`

Description: This functions returns the list of items of a widget. This function is applicable to widget of the classes `XmList`, `XmCommand` and `XmSelectionBox`.

Parameters:
widget Specifies the widget ID.

`get-selected-items` (Function)

Syntax: `(defun get-selected-items (widget))`

Description: This functions returns the list of selected items of a widget. This function is applicable to widget of the classes `XmList`, `XmCommand` and `XmSelectionBox`.

Parameters:
widget Specifies the widget ID.

15.3 The `XmList` Widget Class

The functions described in this section are used to manipulate the item list of an `XmList` widget. Refer to the documentation for details (see [5], pages 326–350).

`list-add-item` (Function)

Syntax: `(defun list-add-item (widget item position))`

Description: This function adds an item to an XmList widget at a given position. If `position` is 0, the item is appended to the item list.

Parameters:
widget Specifies the widget ID.
item Specifies the new item.

CLM function	OSF/Motif function
`list-add-item`	`XmListAddItem()`
`list-add-item-unselected`	`XmListAddItemUnselected()`
`list-delete-item`	`XmListDeleteItem()`
`list-delete-pos`	`XmListDeletePos()`
`list-deselect-all-items`	`XmListDeselectAllItems()`
`list-deselect-item`	`XmListDeselectItem()`
`list-deselect-pos`	`XmListDeselectPos()`
`list-select-item`	`XmListSelectItem()`
`list-select-pos`	`XmListSelectPos()`
`list-set-bottom-item`	`XmListSetBottomItem()`
`list-set-bottom-pos`	`XmListSetBottomPos()`
`list-set-horiz-pos`	`XmListSetHorizPos()`
`list-set-top-item`	`XmListSetItem()`
`list-set-top-pos`	`XmListSetPos()`

Table 15.2: Functions for the `XmList` class

position	Specifies the item position.

`list-add-item-unselected` (Function)

Syntax: `(defun list-add-item-unselected (widget item position))`

Description: This function adds an item to an XmList widget at a given position. If `position` is 0, the item is appended to the item list.

Parameters:

widget	Specifies the widget ID.
item	Specifies the new item.
position	Specifies the item position.

`list-delete-item` (Function)

Syntax: `(defun list-delete-item (widget item))`

Description: This function deletes an item from a list widget.

Parameters:

widget	Specifies the widget ID.
position	Specifies the item.

`list-delete-pos` (Function)

Syntax:	`(defun list-delete-pos (widget item))`	
Description:	This function deletes an item from a list widget at a given position.	
Parameters:	*widget*	Specifies the widget ID.
	position	Specifies the item position.

`list-deselect-all-items` (Function)

Syntax:	`(defun list-deselect-all-items (widget))`	
Description:	This function deselects all items of a list widget.	
Parameters:	*widget*	Specifies the widget ID.

`list-deselect-item` (Function)

Syntax:	`(defun list-deselect-item (widget))`	
Description:	This function deselects an item of a list widget.	
Parameters:	*widget*	Specifies the widget ID.
	item	Specifies the item.

`list-deselect-pos` (Function)

Syntax:	`(defun list-deselect-pos (widget))`	
Description:	This function deselects an item of a list widget at a given position.	
Parameters:	*widget*	Specifies the widget ID.
	position	Specifies the item position.

`list-select-item` (Function)

Syntax:	`(defun list-select-item (widget item))`	
Description:	This function selects an item in a list widget.	
Parameters:	*widget*	Specifies the widget ID.
	item	Specifies the item.

`list-select-pos` (Function)

Syntax: `(defun list-select-pos (widget position))`

Description: This function selects an item at a given position.

Parameters:

widget	Specifies the widget ID.
position	Specifies the item position.

`list-set-bottom-item` (Function)

Syntax: `(defun list-set-bottom-item (widget item))`

Description: This function moves the visible part of the item list and makes the given item the last one to be visible in the list widget.

Parameters:

widget	Specifies the widget ID.
item	Specifies the item.

`list-set-bottom-pos` (Function)

Syntax: `(defun list-set-bottom-pos (widget position))`

Description: This function moves the visible part of the item list and makes the item at the given position the last one to be visible in the list widget.

Parameters:

widget	Specifies the widget ID.
item	Specifies the item position.

`list-set-horiz-pos` (Function)

Syntax: `(defun list-set-horiz-pos (widget position))`

Description: This function moves the horizontal scrollbar of a list widget to the given position.

Parameters:

widget	Specifies the widget ID.
position	Specifies the new horizontal position.

`list-set-top-item` (Function)

Syntax: `(defun list-set-top-item (widget item))`

Description: This function moves the visible part of the item list and makes the given item the first one to be visible in the list widget.

Parameters: *widget* Specifies the widget ID.

item Specifies the item.

`list-set-top-pos` (Function)

Syntax: `(defun list-set-top-pos (widget position))`

Description: This function moves the visible part of the item list and makes the item at the given position the first one to be visible in the list widget.

Parameters: *widget* Specifies the widget ID.

item Specifies the item position.

15.4 Getting Automatically Created Children

A number of Motif widgets automatically create children. The following functions return the widget IDs of these children to make them accessible from CLM. The desired child is denoted by a keyword which is formed from the constants used in the Motif documentation by the usual CLM conventions.

`message-box-get-child` (Function)

Syntax: `(defun message-box-get-child (widget child-name))`

Description: This function returns one of the children of a message-box widget.

Parameters: *widget* Specifies the widget ID.

item Specifies the child-name (e.g. `:ok-button`).

`selection-box-get-child` (Function)

Syntax: `(defun selection-box-get-child (widget child-name))`

Description: This function returns one of the children of a selection-box widget.

Parameters: *widget* Specifies the widget ID.

item Specifies the child-name (e.g. `:ok-button`).

`file-selection-box-get-child` (Function)

Syntax: `(defun file-selection-box-get-child (widget child-name))`

Description: This function returns one of the children of a file-selection-box widget.

Parameters: *widget* Specifies the widget ID.
item Specifies the child-name (e.g. `:ok-button`).

`command-get-child` (Function)

Syntax: `(defun command-get-child (widget child-name))`

Description: This function returns one of the children of a command widget.

Parameters: *widget* Specifies the widget ID.
item Specifies the child-name (e.g. `:ok-button`).

`option-label-gadget` (Function)

Syntax: `(defun message-box-get-child (widget))`

Description: This function returns the label-gadget child of an option-menu.

Parameters: *widget* Specifies the widget ID.

`option-button-child` (Function)

Syntax: `(defun option-button-child (widget))`

Description: This function returns the cascade-button-gadget child of an option-menu.

Parameters: *widget* Specifies the widget ID.

15.5 Setting Child Roles

The following functions specify the roles of the different children of a scrolled-window or main-window.

`scrolled-window-set-areas` (Function)

Syntax:

```
(defun scrolled-window-set-areas (widget
  &key (horizontal-scroll-bar-widget nil)
  (vertical-scroll-bar-widget nil)
  (work-region-widget nil)))
```

Description: This function returns one of the children of a message-box widget.

Parameters: *widget* Specifies the widget ID of the scrolled-window.

main-window-set-areas (Function)

Syntax:

```
(defun main-window-set-areas (widget
      &key (menu-widget nil)
      (command-widget nil)
      (horizontal-scroll-bar-widget nil)
      (vertical-scroll-bar-widget nil)
      (work-region-widget nil)))
```

Description: This function returns one of the children of a message-box widget.

Parameters: *widget* Specifies the widget ID of the main-window.

Appendix A

Changes from Release 2.0 to 2.1

```
- Changed CurrentTime to XtLastTimestampProcessed() in ClmFchangeCursor to
  avoid race conditions.

- Resource converters for accessing enumeration resources were added.
  Additional converters will be easier to define in the future.

- The keywords for the following resources have been changed in order to meet
  the unique naming conventions:

     XmNresizePolicy          ==> :resize-{any,grow,none}
     XmNdefaultButtonType     ==> :dialog-{cancel,ok,help}-button
     XmNfileTypeMask          ==> :file-{regular,directory,any-type}
     XmNcommandWindowLocation ==> :command-{above,below}-workspace
     XmNmultiClick            ==> :multiclick-{keep,discard}
     XmNselectionPolicy       ==> :{single,multiple,extended,browse}-select
     XmNshadowType            ==> :shadow-{in,out,etched-in,etched-out}
     XmNalignment             ==> :alignment-{center,end,beginning}
     XmNdialogStyle           ==> :dialog-{....}
     XmNdialogType            ==> :dialog-{....}

  To avoid many code changes, the old values are still
  available, but might disappear in the future.

- Resource converters for accessing Pixel and Bitmap resources were added,
  as well as some other converters.

- Added various #if's to allow compilation under Motif 1.0. Not all
  features are available under 1.0.

- Timers now have integers and not addresses as IDs.

- New functions added with their opcodes
  (Thanks to Peter Cousseau (Peter.Cousseau@a.nl.cs.cmu.edu)
             Christopher Hoover (ch@lks.csi.com)
             Chris Richardson (cer@Franz.COM)
   for most of these functions and a number of other patches)
    message-box-get-child          99
    selection-box-get-child       108
```

```
    file-selection-box-get-child   93
    command-get-child              90
    option-label-gagdet           101
    option-button-gadget          100
    main-window-set-areas         150
    scrolled-window-set-areas     107
    is-realized                   143
    get-multi-click-time          144
    is-valid-widget-id            146
    widget-full-name              147
    widget-full-class             148
    cascade-button-highlight       87
    last-timestamp-processed      145
    text-set-highlight            128
    text-get-last-position        116
    text-set-selection            118
    text-get-insertion-position   115
    text-replace                  125
    text-get-substring            151
    text-search                   152

- Added the following text functions:
    text-clear-selection
    text-cut
    text-copy
    text-get-baseline
    text-get-selection
    text-get-selection-position
    text-paste
    text-pos-to-xy
    text-remove
    text-scroll
    text-set-add-mode
    text-show-position
    text-xy-to-pos

- Added run-status support for Franz.

- Added uppercasing when sending symbols if Allegro Lisp uses lowercase.

- Added silent operation and some customization flags to td. Renamed td
  to clm-td, toolkits to clm-server and toolkitd to clmd.

- The CLM server can now be started from Lisp.

- Text functions are consistently renamed text-<bla>. Affects the documented
  functions insert-text and append-text.

- A number of events pass more details to their event handler function:
    ClientMessage:          message-type {5/10/20 numbers}
    FocusIn, FocusOut:      mode detail
    ConfigureNotify:        x y width height
    VisibilityNotify:       state
```

```
  PropertyNotify:         atom state time
  SelectionClear:         selection time
  SelectionRequest:       requestor selection target property time
  SelectionNotify:        selection target property time
For an explanation of these values, see the Xlib documentation.
```

- Errors and warnings are now really distinguished. When errors are raised, the execution of the application should probably not be continued (because e.g. a widget could not be created). Implementation bugs are also signalled as errors. Warnings indicate that the requested function could not be performed, but execution may continue.

- An error handler has been added that allows to resume execution after an error without entering the debugger. The question "Restart main loop ?" has been changed into a debugger selection.

Bibliography

[1] Thomas Berlage. *"OSF/Motif: Concepts and Programming"*. Addison Wesley, 1991.

[2] James Gettys, Robert W. Scheifler, and Ron Newman. *"Xlib – C Language X Interface. X Version 11, Release 4"*, 1989.

[3] Kerry Kimbrough and LaMott Oren. *"Common Lisp User Interface Environment"*. Texas Instruments, Sept. 1988. Version 1.15.

[4] Joel McCormack, Paul Asente, and Ralph R. Swick. *"X Toolkit Intrinsics – C Language Interface. X Version 11, Release 4"*, 1989.

[5] Open Software Foundation. *"Motif Programmer's Reference Manual"*, 1990. Version 1.1.

[6] Open Software Foundation. *"Motif Style Guide"*, 1990. Version 1.1.

[7] Open Software Foundation. *"Motif User Interface Language Programmer's Guide"*, 1990. Version 1.1.

[8] Open Software Foundation. *"Motif Window Manager Programmer's Guide"*, 1990. Version 1.1.

[9] Guy L. Steele. *"Common Lisp"*. Digital Press, 1984.

Index