X/Open CAE Specification

X/Open Common Desktop Environment (XCDE)

Definitions and Infrastructure

X/Open Company Ltd.

X/Open CAE Specification

X/Open Common Desktop Environment (XCDE) Definitions and Infrastructure

ISBN: 1-85912-070-9

X/Open Document Number: C324

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

There are two X/Open CAE Specifications (see above) defining the X/Open Common Desktop Environment (XCDE) requirements:

- X/Open Common Desktop Environment — Definitions and Infrastructure (**XCDI**) (this document)
- X/Open Common Desktop Environment — Services and Applications (**XCSA**)

The **XCDI** and **XCSA** documents are mutually dependent specifications, which have been split into two volumes for convenience of use and publication.

The **XCDI** specification provides common definitions for the **XCDI** specification and the **XCSA** specification; therefore, readers should be familiar with the **XCDI** specification before using the **XCSA** specification. (Readers are also expected to be familiar with the X/Open CAE Specification, **System Interface Definitions**, **Issue 4, Version 2**, which contains a number of applicable definitions.)

Structure

The **XCDI** specification is structured as follows:

- Chapter 1 explains the targets for standardisation, requirements for conforming implementations, and general standards terminology.
- Chapter 2 is a glossary of terms.
- Chapter 3 describes general definitions and requirements for XCDE.
- Chapter 4 describes requirements for the underlying window system and additional XCDE interfaces (widgets).
- Chapter 5 describes miscellaneous XCDE desktop services interfaces.
- Chapter 6 describes the XCDE message services interfaces (ToolTalk).
- Chapter 7 describes the XCDE drag and drop interfaces.
- Chapter 8 describes the XCDE data typing interfaces.

- Chapter 9 describes the XCDE execution management interfaces.

Comprehensive references are available in the index.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords and type names. It is also used to identify brackets that surround optional items in syntax, **[ ]**.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — variable names, for example, substitutable argument prototypes
  - — environment variables, which are also shown in capitals
  - — commands or utilities
  - — external variables, such as *errno*
  - — X Window System widgets
  - — functions; these are shown as follows: *name*( ); names without parentheses are either external variables or function family names
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
- Ellipses, . . ., are used to show that additional arguments are optional.
- Syntax and code examples are shown in `fixed width font`. Brackets shown in this font, `[ ]`, are part of the syntax and do *not* indicate optional items.
- Variables within syntax statements are shown in *`italic fixed width font`*.
- The names of virtual keys, such as <*Help*> or <*Insert*> are used as described by the model keyboard section of the **OSF/Motif Style Guide**.

Trade Marks

DEC® is a registered trade mark of Digital Equipment Corporation.

Helvetica® is a registered trade mark of Linotype AG and/or its subsidiaries.

IBM® is a registered trade mark of International Business Machines Corporation.

Motif™ is a trade mark of Open Software Foundation, Inc.

OPEN LOOK® is a registered trademark of Novell, Inc.

Postscript® is a registered trade mark of Adobe Systems Incorporated.

ToolTalk™ is a trade mark of Sun Microsystems, Inc.

UNIX® is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open® is a registered trade mark, and the ‘‘X’’ device is a trade mark, of X/Open Company Ltd.

X Window System™ is a trade mark of the Massachusetts Institute of Technology.

Acknowledgements

X/Open gratefully acknowledges the CDE sponsoring companies who donated the materials for this specification:

- Hewlett-Packard Company
- International Business Machines Corporation
- Novell, Incorporated.
- Sun Microsystems, Incorporated

Referenced Documents

The following documents are referenced in this specification:

ISO C
ISO / IEC 9899: 1990, Information technology — Programming Languages — C.

ISO / IEC 6429: 1992
Information processing — ISO 7-bit and 8-bit coded character sets — Control functions for coded character sets

ISO 8859-1: 1987
Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1

ISO 8879: 1986
Information processing — Text and office systems — Standard Generalised Markup Language (SGML)

ISO / IEC 9070: 1991
Information technology — SGML support facilities — Registration procedures for public text owner identifiers

ANSI X3.64-1979
Additional Controls for Use with the American National Standard Code for Information Interchange

RFC-822
Internet RFC 822, Crocker, D. Standard for the format of ARPA Internet text messages.

MIME RFCs
Internet RFC 1521, N. Borenstein, N. Freed, MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies.

Internet RFC 1522, K. Moore, MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text.

Internet RFC 1590, J. Postel, Media Type Registration Procedure.

Motif Style Guide
Open Software Foundation, OSF / Motif Style Guide, Revision 1.2 (ISBN: 0-13-643123-2).

The following X / Open documents are referenced in this specification.

XBD
X / Open CAE Specification, September 1994, System Interface Definitions, Issue 4, Version 2 (ISBN: 1-859120-36-9, C434).

XSH
X / Open CAE Specification, September 1994, System Interfaces and Headers, Issue 4, Version 2, September 1994 (ISBN: 1-859120-37-7, C435).

XCU
X / Open CAE Specification, Commands and Utilities, Issue 4, Version 2, September 1994 (ISBN: 1-859120-34-2, C436).

XIG
: X/Open Guide, Internationalisation Guide, July 1993 (ISBN: 1-85912-002-4, G304).

XNFS
: X/Open CAE Specification, Protocols for X/Open Interworking, September 1992, (ISBN: i-872630-66-9, C218). This includes description of XDR (Sun Microsystems' External Data Representation standard), which was originally described in Internet RFC 1014.

XPG4
: X/Open Single UNIX Specification (Spec. 1170) — Four Volume Set, September 1994 (ISBN: 1-85912-054-7, T405).

X Protocol
: X/Open CAE Specification, Window Management (X11R5): X Window System Protocol, April 1995 (ISBN: 1-85912-087-3, C507).

Xlib
: X/Open CAE Specification, Window Management (X11R5): Xlib — C Language Binding, April 1995 (ISBN: 1-85912-088-1, C508).

Xt
: X/Open CAE Specification, Window Management (X11R5): X Toolkit Intrinsics, April 1995 (ISBN: 1-85912-089-X, C509).

ICCCM
: X/Open CAE Specification, Window Management (X11R5): File Formats and Application Conventions, April 1995 (ISBN: 1-85912-090-3, C510).

Motif
: X/Open CAE Specification, X/Open Motif Toolkit API, March 1995 (ISBN: 1-85912-024-5, C320).

XCS
: X/Open CAE Specification, Calendaring and Scheduling API (XCS), March 1995 (ISBN: 1-85912-076-8, C321).

XCSA
: X/Open CAE Specification, Common Desktop Environment:— Services and Applications (XCSA), March 1995 (ISBN: 1-85912-074-1, C323).

Chapter 1

Introduction

1.1 Overview

This document describes the X/Open Common Desktop Environment, a common graphical user interface environment supported on systems supporting the X Windows System. The XCDE specification addresses the following standardisation targets:

Application Portability

This target requires that all systems provide identical base interfaces, documenting application program interfaces (APIs), command-line interfaces (CLIs), and data formats. Application portability is the primary goal of this document.

The APIs are defined in terms of the source code interfaces for the C programming language, which is defined in the ISO C standard. It is possible that some implementations may make the interfaces available to languages other than C, but this document does not currently define the source code interfaces for any other language.

The CLIs are defined using the conventions defined in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**, making them accessible from shell scripts or from C source code employing such functions as *system*( ) or *execlp*( ), defined in the X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**.

Some data formats are described in tabular form, others in syntactical description languages such as BNF. The method chosen depends on the complexity of the format.

In some cases, applications can access services via an ‘‘actions interface,’’ as described in Chapter 9 on page 467. Both APIs and CLIs are provided to actions.

This document allows an application to be built using a set of services that are consistent across all systems that conform to this specification (see Section 1.2 on page 2). Such systems are termed XCDE-conformant systems. Applications written in C or as shell scripts using only these interfaces and avoiding implementation-dependent constructs are portable to all XCDE-conformant systems.

User Portability

This target requires that all systems provide identical base functionality with similar look and feel or driveability.

This document attempts to provide adequate user portability by balancing two competing priorities:

- The desire for completely identical user interfaces, down to the exact menu structure, button placement, and pixel arrangement.
- The desire for flexibility in the interfaces to allow for innovation in future versions or competing implementations.

This document achieves this balance by specifying or referring to detailed style guidelines for system services (the same as those recommended for applications) and providing detailed ‘‘capability lists’’ that describe the features to be supported

by an interactive service. The result is that a reasonably competent user would feel comfortable and productive moving between competing XCDE-conformant implementations.

System Interoperability
: This target requires that all systems intercommunicate using common data formats and standardised protocols. This version of this document concentrates on the former, but includes protocols when they have achieved the level of stability comparable to the APIs and suitable base documentation is available. Note that this situation is similar to that of XPG4, which does not attempt to document all formats and protocols for system interoperability. Future versions of the XCDE specification should provide additional system interoperability details.

Component Portability / Replaceability
: This target requires that the interfaces between services and service providers be standardised so that an Independent Software vendor (ISV) can market portable replacements for major end-user-visible components, such as the mail tool or calendar manager. This target can be considered a special case of Application Portability as long as there are adequate interfaces to provide comparable end-user services; given this, the mail tool or calendar manager are themselves applications.

This document defines all of its stated interfaces and their run-time behaviour without imposing any particular restrictions on the way in which the interfaces are implemented.

The following areas are outside the scope of this document:

Binary Portability
: This target was avoided because the binary formats of compiled or linked applications are system-dependent.

System Administration
: The details of the installation, maintenance, and performance tuning of the XCDE itself are also system-dependent.

1.2 Conformance

An implementation conforming to the X/Open Common Desktop Environment (the combination of this document and the **XCSA** specification) shall meet the following criteria:

- The system shall support all the interfaces defined within the X/Open Common Desktop Environment, including all:
  - C-language functions and headers
  - Command-line interfaces
  - Action interfaces
  - Inter-system protocols
  - Externally stored or transmitted data formats
- The system may provide additional or enhanced interfaces, headers and facilities not required by the X/Open Common Desktop Environment, provided that such additions or enhancements do not affect the behaviour of a conforming application.

1.3 Terminology

The following terms are used in this specification:

can
This describes a permissible optional feature or behaviour available to the user or application; all systems support such features or behaviour as mandatory requirements.

implementation-dependent
The value or behaviour is not consistent across all implementations. The provider of an implementation normally documents the requirements for correct program construction and correct data in the use of that value or behaviour. When the value or behaviour in the implementation is designed to be variable or customisable on each instantiation of the system, the provider of the implementation normally documents the nature and permissible ranges of this variation. Applications that are intended to be portable must not rely on implementation-dependent values or behaviour.

may
With respect to implementations, the feature or behaviour is optional. Applications should not rely on the existence of the feature. To avoid ambiguity, the reverse sense of *may* is expressed as *need not*, instead of *may not*.

must
This describes a requirement on the application or user.

obsolescent
Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this document. They are retained in this version because of their widespread use. Their use in new applications is discouraged.

should
With respect to implementations, the feature is recommended, but it is not mandatory. Applications should not rely on the existence of the feature.

With respect to users or applications, the word means recommended programming practice that is necessary for maximum portability.

undefined
A value or behaviour is undefined if this document imposes no portability requirements on applications for erroneous program constructs or erroneous data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application using such behaviour is not fully portable to all systems.

unspecified
A value or behaviour is unspecified if this document imposes no portability requirements on applications for correct program construct or correct data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application requiring a specific behaviour, rather than tolerating any behaviour when using that functionality, is not fully portable to all systems.

1.4 Format of Entries

1.4.1 C-Language Functions and Headers

The entries for C-language functions and headers are based on a common format.

NAME
This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS
This section summarises the use of the entry being described. If it is necessary to include a header to use this interface, the names of such headers are shown, for example:

```
#include <stdio.h>
```

DESCRIPTION
This section describes the functionality of the interface or header.

RETURN VALUE
This section indicates the possible return values, if any.

If the implementation can detect errors, ''successful completion'' means that no error has been detected during execution of the function. If the implementation does detect an error, the error will be indicated.

For functions where no errors are defined, ''successful completion'' means that if the implementation checks for errors, no error has been detected. If the implementation can detect errors, and an error is detected, the indicated return value will be returned. No function in this document affects the value of the *errno* variable described in the X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**.

APPLICATION USAGE
This section gives warnings and advice to application writers about the entry.

EXAMPLES
This section gives examples of usage, where appropriate.

FUTURE DIRECTIONS
This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

SEE ALSO
This section gives references to related information.

CHANGE HISTORY
This section shows the derivation of the entry and any significant changes that have been made to it.

The only sections relating to conformance are the **SYNOPSIS**, **DESCRIPTION** and **RETURN VALUE** sections.

1.4.2 Messages

The ToolTalk desktop message set (see Section 6.6 on page 364) and media exchange message set (see Section 6.6.2 on page 396) descriptions use a modified version of the **SYNOPSIS** notation used for C-language functions. Despite the similarity of its appearance to a C-language function prototype, it represents a message created by the *tt_create_message*( ) function (or one of the related message functions), and its arguments are typically added using separate calls to functions such as *tt_message_arg_ival_set*( ). Within the synopsis, the square brackets (**[ ]**) surround optional arguments.

1.4.3 Service Interfaces

In addition to C-language APIs, this document describes services available to the user and to applications. Such services are often represented by a specific utility program, but are sometimes described in more general terms. Services are described using the following categories, expressed in order of increasing specificity:

Capabilities
: A service is required to provide all of the capabilities within a bulleted list. These general capabilities are expressed in a manner that promotes user portability without sacrificing the ability of implementors to innovate by providing additional or improved interfaces.

Actions
: An application can generally access a XCDE service through an action interface; see Chapter 9 on page 467. When available, the action interface is the preferred means for a XCDE application to access a service. Actions are described in a format identical to that used for C-language functions; see Section 1.4.1 on page 4.

Messages
: An application can generally access a XCDE service by sending ToolTalk messages to it and receiving messages in response. A list is presented of the ToolTalk messages that are sent by, or can be received by, the service. (These ToolTalk messages are described in Section 6.6 on page 364 and Section 6.6.2 on page 396).

Command-Line Interfaces
: A service may include one or more utilities. Some utilities are described with a combination of a Capability List (described earlier) and a CLI. The CLI is a subset of the sections described under *Utility Descriptions*.

Utility Descriptions
: Some utilities are expected to be used primarily by portable shell scripts and are presented using the full template introduced by X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**, enhanced as follows:

NAME
: This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS
: The **SYNOPSIS** section summarises the syntax of the calling sequence for the utility, including options, option-arguments and operands.

DESCRIPTION
: The **DESCRIPTION** section describes the actions of the utility. If the utility has a very complex set of subcommands or its own procedural language, an **EXTENDED DESCRIPTION** section is also provided.

Most explanations of optional functionality are omitted here, as they are usually explained in the **OPTIONS** section.

OPTIONS

The **OPTIONS** section describes the utility options and option-arguments, and how they modify the actions of the utility.

Default Behaviour: When this section is listed as ''None'', it means that the implementation need not support any options.

OPERANDS

The **OPERANDS** section describes the utility operands, and how they affect the actions of the utility.

Default Behaviour: When this section is listed as ''None'', it means that the implementation need not support any operands.

STDIN

The **STDIN** section describes the standard input of the utility. This section is frequently merely a reference to the following section, as many utilities treat standard input and input files in the same manner. Unless otherwise stated, all restrictions described in **INPUT FILES** apply to this section as well.

Default Behaviour: When this section is listed as ''Not used,'' it means that the standard input will not be read when the utility is used as described by this document.

INPUT FILES

The **INPUT FILES** section describes the files, other than the standard input, used as input by the utility. It includes files named as operands and option-arguments as well as other files that are referred to, such as startup and initialisation files, databases, etc. Commonly-used files are generally described in one place and cross-referenced by other utilities.

Record formats are described in a notation similar to that used by the C-language function, *printf*( ). See X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2** for a description of this notation.

Default Behaviour: When this section is listed as ''None'', it means that no input files are required to be supplied when the utility is used as described by this document.

RESOURCES

This section, which has no corresponding section in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**, lists the X Window System resources that affect the operation of the utility.

ENVIRONMENT VARIABLES

The **ENVIRONMENT VARIABLES** section lists what variables affect the utility's execution.

Default Behaviour: When this section is listed as ''None'', it means that the behaviour of the utility is not directly affected by environment variables described by this document when the utility is used as described by this document.

ASYNCHRONOUS EVENTS

The **ASYNCHRONOUS EVENTS** section lists how the utility reacts to such events as signals and what signals are caught.

Default Behaviour: When this section is listed as ''Default'', or it refers to ''the standard action for all other signals,'' it means that the action taken as a result of the signal is one of the following:

1. The action is that inherited from the parent according to the rules of inheritance of signal actions defined in the X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**.
2. When no action has been taken to change the default, the default action is that specified by the X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**.
3. The result of the utility's execution is as if default actions had been taken.

A utility is permitted to catch a signal, perform some additional processing (such as deleting temporary files), restore the default signal action (or action inherited from the parent process) and resignal itself.

STDOUT

The **STDOUT** section describes the standard output of the utility.

Default Behaviour: When this section is listed as ''Not used'', it means that the standard output will not be written when the utility is used as described by this document.

STDERR

The **STDERR** section describes the standard error output of the utility. Only those messages that are purposely sent by the utility are described.

Default Behaviour: When this section is listed as ''Used only for diagnostic messages,'' it means that, unless otherwise stated, the diagnostic messages are sent to the standard error only when the exit status is non-zero and the utility is used as described by this document.

When this section is listed as ''Not used'', it means that the standard error will not be used when the utility is used as described in this document.

OUTPUT FILES

The **OUTPUT FILES** section describes the files created or modified by the utility.

Record formats are described in a notation similar to that used by the C-language function, *printf*( ). See the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2** for a description of this notation.

Default Behaviour: When this section is listed as ''None'', it means that no files are created or modified as a consequence of direct action on the part of the utility when the utility is used as described by this document. However, the utility may create or modify system files, such as log files, that are outside the utility's normal execution environment.

EXTENDED DESCRIPTION

The **EXTENDED DESCRIPTION** section provides a place for describing the actions of very complicated utilities, such as text editors or language processors, which typically have elaborate command languages.

Default Behaviour: When this section is listed as ''None'', no further description is necessary.

EXIT STATUS

The **EXIT STATUS** section describes the values the utility will return to the calling program, or shell, and the conditions that cause these values to be returned. Usually, utilities return zero for successful completion and values greater than zero for various error conditions. If specific numeric values are listed in this section, the system will use those values for the errors described. In some cases, status values are listed more loosely, such as ''>0''. A portable application cannot rely on any specific value in the range shown and must be prepared to receive any value in the range.

CONSEQUENCES OF ERRORS

The **CONSEQUENCES OF ERRORS** section describes the effects on the environment, file systems, process state, and so on, when error conditions occur. It does not describe error messages produced or exit status values used.

When a utility encounters an error condition several actions are possible, depending on the severity of the error and the state of the utility. Included in the possible actions of various utilities are: deletion of temporary or intermediate work files; deletion of incomplete files; validity checking of the file system or directory.

Default Behaviour: When this section is listed as ''Default'', it means that any changes to the environment are unspecified.

APPLICATION USAGE

The **APPLICATION USAGE** section gives advice to the application programmer or user about the way the utility should be used.

EXAMPLES

The **EXAMPLES** section gives one or more examples of usage, where appropriate.

FUTURE DIRECTIONS

The **FUTURE DIRECTIONS** section should be used as a guide to current thinking; there is not necessarily a commitment to implement all of these future directions in their entirety.

SEE ALSO

The **SEE ALSO** section lists related entries.

CHANGE HISTORY

This section shows the derivation of the entry and any significant changes that have been made to it.

In this list, all sections other than **APPLICATION USAGE**, **EXAMPLES**, **FUTURE DIRECTIONS**, **SEE ALSO** and **CHANGE HISTORY** are related to conformance.

Chapter 2

Glossary

2.1 Terms Defined by XCDE

The following terms are used in this document:

abandoned action
(Execution Management Services) An action that is no longer managed by the execution service.

action label
(Execution Management Services) A localised string that provides a textual identification of the action to the user.

action
(Execution Management Services) A desktop construct that provides a method for running applications, sending messages and executing commands.

actions and data types database
(Execution Management, Data Typing) The text files used to control the XCDE data typing (Chapter 8 on page 427) and execution management (Chapter 9 on page 467) services.

actions table
(Execution Management Services) The portion of the actions and data types database that describes actions.

attachment
(Mail Services) An encapsulated data object inside a document. In the XCDE mail services, an attachment is a data object within an electronic mail message that is displayed as an icon in the Attachments list. An attachment can be text, sound or graphic. Multiple messages can be added (attached) to a single email message. An attached message is displayed or activated by selecting it with the mouse button.

auto wraparound
(Terminal Emulation Services) A special mode in which text automatically moves to the next line when it reaches the right margin.

auto-repeat key
(Terminal Emulation Services) A keyboard character that keeps repeating as long as the key is pressed.

backdrop
(Workspace Management Services) A background the user selects via the style manager to make the workspace visually distinctive.

base height
(Terminal Emulation Services) The height of the terminal window before accounting for the total height of the lines of text to be displayed.

base width
(Terminal Emulation Services) The width of the terminal window before accounting for the total width of the columns of text to be displayed.

bell
(Terminal Emulation Services) An indicator that a special event has taken place. It can be an audible beep from the computer or a visual indicator.

blanking mode
(Terminal Emulation Services) A special mode for terminal emulator windows to avoid visual clutter. In this mode the pointer is made invisible if it remains stationary for a period of time. The pointer re-appears when it is moved (usually by a mouse) and disappears after a selectable number of seconds or when keyboard input begins.

bounding box
(Terminal Emulation Services) A rectangle used as a guide when designing fonts. Characters in a font are designed to fit generally inside their font's bounding box.

buffer argument
(Execution Management Services) An argument that specifies an object in memory. This contrasts with a file argument, which specifies a file generally located on disk.

category
(ToolTalk) Attribute of a pattern that indicates whether the application is prepared to handle requests that match the pattern or only observe the requests.

character protection attribute
(Terminal Emulation Services) An attribute that applies to text displayed in a terminal emulator window. Text characters for which this attribute is set cannot be erased.

character-spaced font
(Terminal Emulation Services) A special kind of monospaced font whose characters each fit into the font's bounding box.

client data
(Execution Management Services) Data supplied by the client and associated with a callback function. The meaning of the data depends entirely on the client. Client data is often used by a client to send data that it will receive again later.

colour set
(Workspace Management Services) A set of five colours used to represent a single logical colour in the Motif toolkit. For each background colour (the ''logical'' colour), there are associated top shadow, bottom shadow, foreground and select colours, all generated from the background colour. These associated colours are the mechanism for giving widgets their three-dimensional appearance.

contexts
(ToolTalk) Sets of <*name.value*> explicitly included in both messages and patterns. ToolTalk contexts allow fine-grain matching.

The application can use contexts to associate arbitrary pairs with ToolTalk messages and patterns, and to restrict the set of possible recipients of a message. One common use of the restricted pattern matching provided by ToolTalk context attributes is to create sub-sessions. For example, two different programs could be debugged simultaneously with tools such as a browser, an editor, a debugger, and a configuration manager active for each program. The message and pattern context slots for each set of tools contain different values; the normal ToolTalk pattern matching of these values keep the two sub-sessions separate.

Another use for the restricted pattern matching provided by ToolTalk context attributes is to provide information in environment variables and command-line arguments to tools started by the ToolTalk service.

current session
(Session Management Services) The collection of applications, settings and resources that are currently present on the user's desktop. If a user selects to restore the current session upon login, the desktop will be restored to the same state it was in when that user ended the previous session and logged out.

current workspace
(Workspace Management Services) The workspace whose windows are currently visible to the user.

data attributes
(Data Typing) The attributes that determine user-visible interfaces to file or byte-stream data: a human-readable description of the type, the icon to represent it graphically and the actions that apply to it.

data attributes table
(Data Typing) The portion of the actions and data types database that describes data attributes. See Section 8.4.4 on page 460.

data criteria table
(Data Typing) The portion of the actions and data types database that describes data criteria. See Section 8.4.3 on page 455.

data type
(Data Typing) A name characterising the data in a file or byte vector. A data type is named by the value of the DATA_ATTRIBUTES_NAME field in the matching record of the data criteria table of the actions and data types database.

data types database
(Data Typing) See **actions and data types database** on page 11.

data typing
A method of determining the data attributes of a file or byte vector, based on its name, file permissions, symbolic links and content.

display area
(Help Services) An area in help dialogs that shows a help topic.

drag
(Drag and Drop) A user interaction in which elements or their representations change their position or appearance in conjunction with the movement of the pointer.

drag and drop
(Drag and Drop) A user interaction in which a user drags source elements to a target element on which they are dropped.

drop
(Drag and Drop) A user action that terminates a drag, identifying the destination of the drag and drop interaction as the element under the pointer.

drop zone
(Drag and Drop) An area of the workspace, including the Trash Can, Print Manager and Mailer controls, that accepts a dropped icon. Icons can be dropped on the workspace for quick access.

dynamic message patterns
(ToolTalk) A message pattern provided by the application that is created while the application is running. See *tt_pattern_create*( ) and *tt_pattern_register*( ) in Chapter 6.

deserialise
(ToolTalk) Decode a data structure from an architecturally neutral stream of bytes.

edict
(ToolTalk) A notice that looks like a request. For example, if a request returns no data (or if the sender does not care about the returned data), the application can use an edict to broadcast that request to a set of tools. Since the message is a notice, no data will be returned, no replies will be received, and the sender is not told whether any tool receives the message.

escape character
(Terminal Emulation Services) A character that generally indicates the beginning of a command to be performed, not text to be displayed or passed to the application. The character code in the referenced ISO / IEC 646: 1983 standard is decimal 27.

execution host
(Execution Management Services) A host computer that runs an application invoked by an action. This may be the same computer where the action resides, or it may be another computer on the network.

execution string
(Execution Management Services) A string that specifies the command to invoke.

fail a request
(ToolTalk) Inform a sending application that the requested operation cannot be performed, by calling *tt_message_fail*( ) in Chapter 6. This is a voluntary failure, as opposed to a message not being sendable.

folder
(File Management Services) A representation of a directory in the underlying file system.

front panel
(Front Panel Services) A centrally located window containing controls for accessing applications and utilities. The front panel occupies all workspaces.

general help dialog
(Help Services) A window that displays help information and provides full navigation capabilities. See also **quick help dialog** on page 18, which offers more limited capabilities.

handler
(ToolTalk) The distinguished recipient procid of a message.

handle a request
(ToolTalk) Perform the operation requested by the sending application.

hard reset
(Terminal Emulation Services) An operation that fully restores the terminal emulator to a specific startup state, as defined by the emulation. This includes everything done by a soft reset.

height increment
(Terminal Emulation Services) The height in pixels of a single line of text. It is used to calculate the height of the terminal emulator window

HelpTag
(Help Services) The markup language used for creating XCDE help volumes. HelpTag complies with the referenced SGML standard.

help topic
(Help Services) The smallest addressable piece of help information. It may be authored in HelpTag or system manual-page macros, or be a text file or string in the referenced ISO/IEC 646:1983 standard codeset. Help topics authored in HelpTag may reference additional topics.

help type
(Help Services) The format of the data in a given help topic. Formats include HelpTag topic, string data, system manual page and text file.

help volume
(Help Services) A collection of help topics authored in HelpTag.

home session
(Session Management Services) The collection of applications, settings and resources on the desktop. If a user selects to restore the home session upon login, the desktop is restored to the same initial state every time the user logs in, regardless of its state when the user last logged out.

icon name
(Execution Management Services) The name of a field in the action database. It is generally used by applications to construct the location where the bitmap of the icon is located.

initial session
(ToolTalk) The ToolTalk session in which the application was started.

jump scrolling
(Terminal Emulation Services) A specific scrolling behaviour in which the screen may be scrolled by more than one line at a time.

location ID
(Help Services) The identifier for a particular location in a help topic that can be accessed directly.

login shell
(Terminal Emulation Services) The initial shell invoked when a user logs in. For *ksh* and *sh* users, the login shell startup is controlled by the system **profile** and the user **.profile** files.

mail header
(Mail Services) The lines in a mail message that precede the empty line that marks the beginning of the message text, as described in the referenced RFC-822 specification.

margin bell
(Terminal Emulation Services) An indicator that user input has reached the right margin of the window.

mark
(ToolTalk) An integer that represents a location on the API storage stack; see *tt_mark*( ) and *tt_release*( ) in Chapter 6.

message
(ToolTalk) A structure that the ToolTalk service delivers to processes. A ToolTalk message consists of an operation name, a vector of type arguments, a status value or string pair, and ancillary addressing information.

message callback
(ToolTalk) A client function. The ToolTalk service invokes this function to report information about the specified message back to the sending application; for example, the message failed or the message caused a tool to start.

message pattern
(ToolTalk) Defines the message the application is prepared to receive.

message protocol
(ToolTalk) A set of ToolTalk messages that describe operations the applications agree to perform.

module
(Application Building Services) An arbitrary subdivision of a project; see **project** on page 17. Projects can be subdivided into modules for clarity of architectural structure, for ease of controlling multiple programmers on the same project, and so forth.

monospaced font
(Terminal Emulation Services) A font composed of characters that are all of the same width.

netfilename
(ToolTalk) A canonical form of a pathname that can be passed to other hosts on the network and converted back to a local pathname for the same file. See *tt_file_netfile*( ) in Chapter 6.

notice
(ToolTalk) An informational message used by an application to announce an event. Zero or more tools may receive a given notice. The sender is not informed whether any tools receive its notice because replies cannot be sent for a notice.

object
(File Management Services) A representation of a file in the underlying file system. Examples of objects are text files, actions or directories.

object content
(ToolTalk) A piece, or pieces, of an ordinary file, managed by the application that creates or manages the object; for example, a paragraph, a source code function or a range of spreadsheet cells.

object-oriented messages
(ToolTalk) Messages addressed to objects managed by applications.

object specification (spec)
(ToolTalk) Standard properties of an object, such as the type of object, the name of the file in which the object contents are located, and the object owner.

object type (otype)
(ToolTalk) Addressing information that the ToolTalk service uses when delivering object-oriented messages to an application.

object type identifier (otid)
(ToolTalk) Identifies the object type.

objid
(ToolTalk) The identifier for a ToolTalk object. The same ToolTalk object can have different objids as it is moved from one file system to another; see *tt_objid_equal*( ) in Chapter 6. Objids are returned by *tt_spec_create*( ).

observe a message
(ToolTalk) View a message without performing any operation that may be requested.

observe promise
(ToolTalk) Guarantee that the ToolTalk service will deliver a copy of each matching message to ptypes with an observer signature of start or queue disposition. The ToolTalk service will deliver the message either to a running instance of the ptype, by starting an instance, or by queueing the message for the ptype.

opaque
(ToolTalk, Terminal Emulation Services) A value or structure that has meaning only when passed through a particular interface.

opname (op)
(ToolTalk) The name of the message.

opnum
(ToolTalk) A mechanism for indicating the static pattern that caused the message to be received. Opnums and opnum callbacks are to static patterns as pattern callbacks are to dynamic patterns.

option argument keyword
(Execution Management Services) A special symbolic name that refers to a value determined at run time. For example, **%Arg_1%** refers to the first argument passed to *DtActionInvoke*( ).

option string
(Execution Management Services) A string containing command-line options to an execution string.

overstriking
(Terminal Emulation Services) Rendering the same characters several times at close to the same position. This technique originated to emulate bold fonts on a typewriter.

palette object
(Application Building Services) A user interface object that is included in an application by dragging it from the object palette of the application building services.

paragraph
(Text Editing Services) One or more words preceded by a blank line or the beginning of the text and followed by a blank line or the end of the text.

pattern callback
(ToolTalk) A client function. The ToolTalk service invokes this function when a message is received that matches the specified pattern.

pixel offset
(Terminal Emulation Services) The distance in pixels between two locations.

procedural message
(ToolTalk) A message addressed by operation name.

procid
(ToolTalk) The process identifier. The procid is a principal that can send and receive ToolTalk messages. It is an identity, created and passed by the ToolTalk service, that a process must assume to send and receive messages. A single operating system process can use multiple procids. A procid is the value returned by *tt_open*( ) in Chapter 6.

project
(Application Building Services) An application under construction by the application building services. A project can consist of zero or more modules.

proportional font
(Terminal Emulation Services) A font whose characters are various widths.

pseudo-terminal
(Terminal Emulation Services) An implementation-dependent driver that simulates the action of a terminal device; a software abstraction of the communications path between the system and a

terminal. A pseudo-terminal connection can be used between an application or shell and a terminal emulation window.

ptid
(ToolTalk) The process type identifier.

ptype
(ToolTalk) The process type.

quick help dialog
(Help Services) A window that displays a help topic. See also **general help dialog** on page 14, which offers additional capabilities.

registration context
(Workspace Management Services) A handle that uniquely identifies a callback function. A registration context is returned when a workspace callback function is added. It must be saved if the callback function is to be removed.

reject a request
(ToolTalk) Tell the ToolTalk service, using *tt_message_reject*( ) in Chapter 6, that the receiving application is unable to perform the requested operation and that the message should be given to another tool.

reparenting window manager
(Terminal Emulation Services) Applications create toplevel windows that they request to be children of the root window. A reparenting window manager makes toplevel windows children of its windows instead. The window manager windows generally act as a frame around the application's toplevel window and contain such elements as resize handles and title bars.

request
(ToolTalk) A call for an action. The results of the action are recorded in the message, and the message is returned to the sender as a reply. A request asks that an operation be performed. It has a distinguished recipient (handler) responsible for performing the indicated operation.

reverse wraparound
(Terminal Emulation Services) The cursor movement that wraps the cursor up to the previous line if the cursor is located at the left margin and the user presses backspace.

rpc.ttdbserverd
(ToolTalk) The ToolTalk database server process.

scope
(ToolTalk) The attribute of a message or pattern that determines how widely the ToolTalk service looks for matching messages or patterns.

selection extension
(Terminal Emulation Services) An increase in the amount of text selected.

serialise
(ToolTalk) Encode a data structure into an architecturally neutral stream of bytes.

sessid
(ToolTalk) The identifier for a ToolTalk session. It is an opaque character string.

session
(ToolTalk) A group of ToolTalk processes that are related either by being in the same desktop or the same process tree.

signal handler
(Terminal Emulation Services) A software routine that deals with signals the application may receive.

signature
(ToolTalk) A pattern in a ptype or otype. A signature can contain values for disposition and operation numbers.

- A ptype signature (*psignature*) describes the procedural messages that the application is prepared to receive.
- An otype signature (*osignature*) defines the messages that can be addressed to objects of the type.

slave device
(Terminal Emulation Services) A system device that provides to application processes a terminal device interface as described in the General Terminal Interface (*termios*) of the X/Open CAE Specification, **System Interface Definitions**, **Issue 4, Version 2**.

slotname
(ToolTalk) The name of a ToolTalk context. See **contexts** on page 12.

soft reset
(Terminal Emulation Services) An operation that partially restores the terminal emulator to a specific startup state, as defined by the emulation. It is a subset of a hard reset.

source
(Drag and Drop) The object that is selected, dragged and dropped in a drag-and-drop action.

source indicator
(Drag and Drop) The part of the pointer displayed during drag and drop that describes the source.

spec
(ToolTalk) See **object specification (spec)** on page 16.

starting shell
(Terminal Emulation Services) The initial shell controlling interaction when the terminal window appears.

state indicator
(Drag and Drop) The part of the pointer displayed during drag and drop that indicates whether the pointer is at a place where a drop is likely to result in a successful operation.

static message pattern
(ToolTalk) A signature. See **signature**.

subpanel
(Front Panel Services) A component associated with a control on the front panel. It provides additional elements associated with that control.

terminal emulator
(Terminal Emulation Services) A window that emulates a particular type of terminal for running non-window programs. Terminal emulator windows are most commonly used for typing commands to interact with the computer's operating system.

text rendering
(Terminal Emulation Services) Drawing text on a video screen.

tool
(ToolTalk) An application or utility that can be manipulated using ToolTalk services.

ToolTalk API
(ToolTalk) The functions and headers in Chapter 6.

ToolTalk service
(ToolTalk) The implementation of the ToolTalk API and components.

ToolTalk types database
(ToolTalk) The database that stores ToolTalk type information.

topic tree
(Help Services) An area in the general help dialog that aids navigation of topics that can be browsed in the current help volume.

underscore
(Terminal Emulation Services) The character ‘‘_’’.

value type (vtype)
(ToolTalk) An application-defined string that describes what kind of data a message argument contains. The ToolTalk service only uses vtypes to match sent message instances with registered message patterns.

virtual keys
(Terminal Emulation Services) Logical key translations, as defined by the X/Open CAE Specification, **Motif Toolkit API**.

width increment
(Terminal Emulation Services) The width in pixels of a single column of text. It is used to calculate the width of the terminal emulator window.

window menu
(Workspace Management Services) The menu displayed by choosing the Window menu button. The menu provides choices that manipulate the location or size of the window, such as Move, Size, Minimise and Maximise.

word
(Text Editing Services) One or more non-white-space characters preceded by the beginning of the text, the beginning of a line, or a white-space character and trailed by the end of the text, the end of a line, or a white-space character.

workspace
(Workspace Management Services) A ‘‘virtual screen’’ that contains a set of windows. Workspaces provide a way of grouping together logically related windows. Each workspace is shown independent of the other workspaces so that only those windows related to the immediate task are visible. Workspaces organise windows by task and efficiently use screen space.

workspace identifier
(Workspace Management Services) An X atom that indicates the name of the workspace.

workspace functions
(Workspace Management Services) Capabilities that appear in a window menu to allow the user to specify which workspace a window occupies.

workspace manager
(Workspace Management Services) A program that implements workspaces.

workspace name
(Workspace Management Services) The resource name for a workspace. This name is converted to an X atom and used in the workspace management API. The workspace name is generated dynamically by the workspace manager. Since it is a resource name, the characters are limited to those in the X Portable Character Set. (See **workspace title**.)

workspace title
(Workspace Management Services) The user-visible title associated with a workspace. It is displayed in the workspace button in the front panel. It is interpreted in the locale in which the workspace manager is running. The user can change the workspace title.

XNFS
The Protocols for X/Open Interworking (X/Open CAE Specification) includes description of ToolTalk, to serialise data for transmission using the External Data Representation data description language and data representation protocol; this was originally described in RFC 1014.

2.2 Terms From Other Standards

The following terms used in this document are defined in X/Open CAE Specification, **System Interface Definitions**, **Issue 4, Version 2**:

absolute pathname
appropriate privileges
argument
assignment
asterisk
backquote
backslash
backspace character
basename
blank character
blank line
block special file
byte
carriage-return character
character
character special file
character string
child process
codeset
collation
collation sequence
command
command language interpreter
current working directory
cursor position
device
directory
directory entry (link)
dot
dot-dot
double-quote
effective group ID
effective user ID
empty directory
empty line
empty string (null string)
executable file
file
file access permissions
file descriptor
file hierarchy
file mode
file permission bits
file system
file type
filename
form-feed character
group ID
group name
home directory
internationalisation
line
locale
localisation
login
login name
message catalogue
mode
mount point
negative response
newline character
NUL
null byte
null pointer
null string
operand
operator
option
option-argument
parameter
parent directory
parent process
path prefix
pathname
pathname component
pathname resolution
period
permissions
pipe
positional parameter
privilege
process
program
radix character
redirection
regular expression
regular file
relative pathname
root directory
scroll
shell
shell script
signal
slash
space character
standard error
standard input
standard output
stream
string
system
tab character
terminal
terminal device
text file
user ID
user name
utility
white space
working directory

The following terms used in this document are defined in the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**, the X/Open CAE Specification, **Window Management: Xlib — C Language Binding**, and the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

access control list
active grab
ancestors
API
application
application programmer
atom
background
backing store
BDF
bit gravity
bit plane
bitmap
border
button grabbing
byte order
children
class
client
clipping region
colormap
compound text
connection
containment
coordinate system
cursor
depth
device
directColor
display
drawable
event
event mask
event propagation
event source
event synchronisation
exposure event
extension
focus window
font
gadget
GContext
glyph
grab
graphics context
gravity
grayScale
GUI
hotspot
ICCCM
identifier
inferiors
input focus
input manager
InputOnly window
InputOutput window
instance
key grabbing
keyboard grabbing
KEYSYM
mapped
method
modifier keys
monochrome
object
obscure
occlude
padding
parent window
passive grab
pixel
pixel value
pixmap
plane
plane mask
pointer
pointer grabbing
pointing device
policy
property
property list
pseudoColor
redirecting control
reply
request
resource
RGB values
root
root window
save set
scanline
scanline order
screen
selection
server
server grabbing
session manager
sibling
stacking order
staticColor
staticGray
stipple
string equivalence
style guide
tile
timestamp
TrueColor
type
UI platform
UI specification
UIDL
user
user interface
viewable
visible
widget
widget programmer
window gravity
window manager
X Protocol
XLFD
Xlib
Xt Intrinsics
XYFormat
ZFormat

The following terms used in this document are defined in the X/Open CAE Specification, **Motif Toolkit API**:

atom	icon	pointer	scrollbar
border	menu	protocol	widget
class	Motif	Resource	window
compound text	object	scroll	window manager
drag and drop			

Chapter 3

General Definitions and Requirements

3.1 XCDE Data Format Naming

The X/Open Common Desktop Environment supports a common name space to describe file formats that can be interchanged between applications—the Media name space. The names in the Media name space describe the form of the data itself. Media names are used as ICCCM selection targets (see the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**); they are named in the MEDIA field of data type records (see Chapter 8 on page 427); and they are used in the *type* parameter of Media Exchange messages (see Section 6.6.2 on page 396).

The Media name space is a subset of the space of selection target atoms as defined by the ICCCM. All selection targets that specify a data format are valid Media names, and all valid Media names can be used directly as selection targets. Some selection targets specify an attribute of the selection (for example, LIST_LENGTH) or a side effect to occur (for example, DELETE), rather than a data format. These selection targets are not part of the Media name space.

The space of Media names follows the ICCCM conventions for selection target atoms. Names typically are in all uppercase, with words separated by underscores. Media names should follow this convention where possible.

The following Media names have the meanings indicated on all XCDE systems. It is implementation dependent whether facilities are provided to manipulate any of these formats.

ADOBE_PORTABLE_DOCUMENT_FORMAT
: Adobe Systems, Incorporated. *Portable Document Format Reference Manual.* Addison-Wesley, ISBN 0-201-62628-4.

APPLE_PICT
: Apple Computer, Incorporated. *Inside Macintosh, Volume V.* Chapter 4, ‘‘Color QuickDraw,’’ Color Picture Format. ISBN 0-201-17719-6.

COMPOUND_TEXT
: Scheifler, Robert W. *Compound Text Encoding, Version 1.1.* MIT X Consortium Standard. X Version 11, Release 5, 1991.

ENCAPSULATED_POSTSCRIPT
ENCAPSULATED_POSTSCRIPT_INTERCHANGE
: Adobe Systems, Incorporated. *PostScript Language Reference Manual.* Appendix H. Addison-Wesley, ISBN 0-201-18127-4.

GIF87 Graphics Interchange Format. June 15, 1987. CompuServe, Incorporated, 1987.

: CompuServe, Incorporated
5000 Arlington Centre Blvd.
Columbus, OH 43220

GIF89 Graphics Interchange Format, Version 89a. 31 July 1990. CompuServe, Incorporated, 1987-1990.

JFIF Hamilton, Eric. *JPEG File Interchange Format, Version 1.02.* September 1, 1992.

C-Cube Microsystems
1778 McCarthy Blvd.
Milpitas, CA 95035

POSTSCRIPT
Adobe Systems, Incorporated. *PostScript Language Reference Manual.* Appendix H. Addison-Wesley, ISBN 0-201-18127-4.

RFC_822_MESSAGE
A mail message formatted in accordance with the referenced RFC-822 and the referenced MIME RFCs.

SND Apple Computer, Incorporated. *Inside Macintosh, Volume V.* Chapter 27, ''Sound Resources.'' ISBN 0-201-17719-6.

STRING Text encoded in the printable characters, plus <tab> and <newline>, of the Latin-1 codeset (see the referenced ISO/IEC 8859-1: 1987 standard)

SUN_AUDIO_DATA
Solaris 2.3 SUNBIN CD-ROM part number 258-3779. Sun Microsystems, Inc., 1993. Audio demo (SUNWaudmo) package: manual pages *audio_intro*(3) and *audio_filehdr*(3).

SUN_RASTER
OpenWindows 3.3 Reference Manual Sun Microsystems, Inc., 1990-1993. Manual page *rasterfile*(5).

TIFF TIFF Revision 6.0, June 3, 1992.

Aldus Corporation
411 First Avenue South
Seattle, WA 98104-2871

WAV IBM PC sound file format — *Multimedia Programming Interface and Data Specifications 1.0*, a joint design by IBM Corporation and Microsoft Corporation, August 1991.

Microsoft Corporation
Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399

XBM X/Open CAE Specification, **Window Management: Xlib — C Language Binding**, Section 11.10, ''Manipulating Bitmaps.''

XPM Le Hors, Arnaud. *XPM Manual: The X PixMap Format, Version 3.0.* Groupe Bull, 1990-1991. Contributed Software. X Version 11, Release 5.

Chapter 4

X Windows and Motif

This chapter describes the dependencies on X Windows and Motif as underlying implementations and defines additional widget interfaces that promote portability of historical applications into the XCDE environment.

4.1 X Protocol

The interfaces in this document require an underlying implementation that support the protocols defined in the X/Open CAE Specification, **Window Management: X Window System Protocol**.

4.2 Xlib Library

The interfaces in this document require an underlying implementation that support the interfaces defined in the X/Open CAE Specification, **Window Management: Xlib — C Language Binding**. The version of that document based on X Windows Version 11, Release 5 is required for XCDE.

4.3 Xt Intrinsics

The interfaces in this document require an underlying implementation that support the interfaces defined in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**. The version of that document based on X Windows Version 11, Release 5 is required for XCDE.

4.4 ICCCM, CT, XLFD, BDF

The interfaces in this document require an underlying implementation that support the interfaces defined in the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**. The version of that document based on X Windows Version 11, Release 5 is required for XCDE.

4.5 Motif Libraries

The interfaces in this document require an underlying implementation that support the interfaces defined in the X/Open CAE Specification, **Motif Toolkit API**.

4.6 X Windows and Motif Data Types

The following data types are defined in X11 and Motif headers (see the X/Open CAE Specification, **Window Management: Xlib — C Language Binding**, the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**, and the X/Open CAE Specification, **Motif Toolkit API**):

ArgList	**Pixel**	**XEvent**
Atom	**Pixmap**	**XmFontList**
Boolean	**Position**	**XmString**
Cardinal	**String**	**XmStringDirection**
Colormap	**Time**	**XmStringTable**
Cursor	**Widget**	**XmTextPosition**
Dimension	**WidgetClass**	**XtCallbackList**
Display	**XButtonPressedEvent**	**XtPointer**

4.7 XCDE Widgets

This section defines the widget classes that provide the XCDE-specific widgets to support application portability at the C-language source level.

NAME

DtComboBox — the ComboBox widget class

SYNOPSIS

```
#include <Dt/ComboBox.h>
```

DESCRIPTION

The DtComboBox widget is a combination of a TextField and a List widget that provides a list of valid choices for the TextField. Selecting an item from this list automatically fills in the TextField with that list item.

Widget subclassing is not supported for the DtComboBox widget class.

The resources for the XmList and XmTextField widgets that are created by the DtComboBox are accessible by using the *XtNameToWidget*( ) function. The names of these widgets are ∗**List** and **Text**, respectively. (The ∗**List** notation is required because the List widget is not an immediate descendant of the DtComboBox. See *XtNameToWidget*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**.)

Classes

The DtComboBox widget inherits behaviour and resources from the *Core*, *Composite* and *XmManager* classes.

The class pointer is **dtComboBoxWidgetClass**.

The class name is *DtComboBoxWidget*.

New Resources

The following table defines a set of widget resources used by the application to specify data. The application can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, the application must remove the **DtN** or **DtC** prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, the application must remove the **Dt** prefix and use the remaining letters (in either lower case or upper case, but including any underscores between words). The codes in the access column indicate if the given resource can be set at creation time (C), set by using *XtSetValues*( ) (S), retrieved by using *XtGetValues*( ) (G), or is not applicable (N/A).

DtComboBox Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
DtNmarginHeight	**DtCMarginHeight**	**Dimension**	2	CSG
DtNmarginWidth	**DtCMarginWidth**	**Dimension**	2	CSG
DtNselectedItem	**DtCSelectedItem**	**XmString**	dynamic	CSG
DtNselectedPosition	**DtCSelectedPosition**	**int**	dynamic	CSG
DtNselectionCallback	**DtCCallback**	**XtCallbackList**	NULL	C
DtNcomboBoxType	**DtCComboBoxType**	**unsigned int**	DtDROP_DOWN_LIST	C

DtNmarginHeight

Specifies the number of pixels added between the top and bottom of the text widget and the start of the shadow.

DtNmarginWidth

Specifies the number of pixels added between the right and left sides of the text widget and the start of the shadow.

DtNselectedItem

This resource is passed through to the XmList to set the **XmNselectedItemCount**

and **XmNselectedItems** as the single item in the **XmNitems** that matches this specified **XmString** in the List. Setting both **DtNselectedItem** and **DtNselectedPosition** in the same argument list produces undefined behaviour.

DtNselectedPosition
This resource is passed through to the XmList to set the **XmNselectedItemCount** and **XmNselectedItems** as the single item at this specified position in the List. Setting both **DtNselectedItem** and **DtNselectedPosition** in the same argument list produces undefined behaviour.

DtNselectionCallback
This callback list is issued when an item is selected from the DtComboBox widget's List. The *call_data* structure contains a *DtComboBoxCallbackStruct* with the reason DtCR_SELECT.

DtNcomboBoxType
This resource determines the style of the DtComboBox. There are two choices:

DtDROP_DOWN_COMBO_BOX
Provides an editable text area.

DtDROP_DOWN_LIST
Provides a static text area.

Inherited Resources

The DtComboBox widget inherits behaviour and resources from the following named superclasses. For a complete description of each resource, see the entry in X/Open CAE Specification, **Motif Toolkit API** for that superclass.

XmManager Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	dynamic	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNforeground	**XmCForeground**	**Pixel**	dynamic	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	NULL	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	dynamic	CSG
XmNhighlightPixmap	**XmCHighlightPixmap**	**Pixmap**	dynamic	CSG
XmNinitialFocus	**XmCInitialFocus**	**Widget**	NULL	CSG
XmNnavigationType	**XmCNavigationType**	**XmNavigationType**	dynamic	CSG
XmNshadowThickness	**XmCShadowThickness**	**Dimension**	dynamic	CSG
XmNstringDirection	**XmCStringDirection**	**XmStringDirection**	dynamic	CG
XmNtopShadowColor	**XmCTopShadowColor**	**Pixel**	dynamic	CSG
XmNtopShadowPixmap	**XmCTopShadowPixmap**	**Pixmap**	dynamic	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	dynamic	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	dynamic	CSG
XmNuserData	**XmCUserData**	**XtPointer**	NULL	CSG

Composite Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNchildren	**XmCReadOnly**	**WidgetList**	NULL	G
XmNinsertPosition	**XmCInsertPosition**	**XtOrderProc**	default procedure	CSG
XmNnumChildren	**XmCReadOnly**	**Cardinal**	0	G

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	dynamic	CSG
XmNancestorSensitive	**XmCSensitive**	**Boolean**	dynamic	G
XmNbackground	**XmCBackground**	**Pixel**	dynamic	CSG
XmNbackgroundPixmap	**XmCPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	XtDefaultForeground	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	dynamic	CG
XmNdepth	**XmCDepth**	**int**	dynamic	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	NULL	C
XmNheight	**XmCHeight**	**Dimension**	dynamic	CSG
XmNinitial-ResourcesPersistent	**XmCInitial-ResourcesPersistent**	**Boolean**	True	C
XmNmappedWhen-Managed	**XmCMappedWhen-Managed**	**Boolean**	True	CSG
XmNscreen	**XmCScreen**	**Screen** *	dynamic	CG
XmNsensitive	**XmCSensitive**	**Boolean**	True	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	dynamic	CSG
XmNwidth	**XmCWidth**	**Dimension**	dynamic	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Callback Information

A pointer to the following structure is passed to each DtComboBox callback:

```
typedef struct {
        int             reason;
        XEvent          *event;
        XmString        item_or_text;
        int             item_position;
} DtComboBoxCallbackStruct;
```

The *reason* argument indicates why the callback was invoked; it is always DtCR_SELECT.

The *event* argument points to the **XEvent** that triggered the callback, or NULL if the callback was not triggered by an **XEvent**.

The *item_or_text* argument is the contents of the Text widget at the time the event caused the callback to be invoked. This data is only valid within the scope of the *call_data* structure, so the application must copy it when it is used outside of this scope.

The *item_position* argument is the new value of the **DtNselectedPosition** resource in the DtComboBox's List. If this is zero, it means the user entered a value in the XmTextField widget.

SEE ALSO

DtComboBoxAddItem( ), *DtComboBoxDeletePos*( ), *DtComboBoxSelectItem*( ), *DtComboBoxSetItem*( ), *DtCreateComboBox*( ); *Composite*, *Constraint*, *Core*, *XmList*, *XmManager*, *XmText*, *XmTextField* in the X/Open CAE Specification, **Motif Toolkit API**; *XtGetValues*( ), *XtSetValues*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

NAME

DtMenuButton — the MenuButton widget class

SYNOPSIS

```
#include <Dt/MenuButton.h>
```

DESCRIPTION

The DtMenuButton widget is a command widget that complements the menu cascading functionality of an XmCascadeButton widget. As a complement to the XmCascadeButton widget, DtMenuButton can only be instantiated outside a MenuPane; the application must use XmCascadeButton widget inside a MenuPane.

The DtMenuButton widget belongs to a subclass of the XmLabel class. Visually, the DtMenuButton widget consists of a label string and a menu glyph. The menu glyph always appears on the right end of the widget and, by default, is a downward pointing arrow.

The DtMenuButton widget has an implicitly created submenu attached to it. The submenu is a popup MenuPane and has this DtMenuButton widget as its parent. The name of the implicitly created submenu is obtained by adding **submenu_** as a prefix to the name of this DtMenuButton widget. The widget ID of the submenu can be obtained by doing an *XtGetValues*( ) on the **DtNsubMenuId** resource of this DtMenuButton widget. The implicitly created submenu must not be destroyed by the user of this widget.

The submenu can be popped up by pressing the menu post Button (see the **XmNmenuPost** resource of the XmRowColumn widget) anywhere on the DtMenuButton widget.

Widget subclassing is not supported for the DtMenuButton widget class.

Classes

The DtMenuButton widget inherits behaviour and resources from the *Core*, *Composite*, *XmPrimitive* and *XmLabel* classes.

The class pointer is **dtMenuButtonWidgetClass**.

The class name is *DtMenuButtonWidget*.

New Resources

The following table defines a set of widget resources used by the application to specify data. The application can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, the application must remove the **DtN** or **DtC** prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, the application must remove the **Dt** prefix and use the remaining letters (in either lower case or upper case, but including any underscores between words). The codes in the access column indicate if the given resource can be set at creation time (C), set by using *XtSetValues*( ) (S), retrieved by using *XtGetValues*( ) (G), or is not applicable (N/A).

DtMenuButton Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
DtNcascadingCallback	**DtCCallback**	**XtCallbackList**	NULL	C
DtNcascadePixmap	**DtCPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
DtNsubMenuId	**DtCMenuWidget**	**Widget**	NULL	SG

DtNcascadingCallback
Specifies the list of callbacks that is called before the popping up of the attached submenu. The reason for the callback is DtCR_CASCADING.

DtNcascadePixmap
Specifies the pixmap displayed as the menu glyph. If no pixmap is specified, a downward pointing arrow is displayed.

DtNsubMenuId
Specifies the widget ID of the popup MenuPane to be associated with this DtMenuButton widget. The popup MenuPane must be created with this DtMenuButton as its parent. This resource cannot be specified at the time of widget creation. The implicit submenu is automatically destroyed by DtMenuButton when this resource is set.

Inherited Resources

The DtMenuButton widget inherits behaviour and resources from the following named superclasses. For a complete description of each resource, see the entry in X/Open CAE Specification, **Motif Toolkit API** for that superclass.

XmLabel Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerator	**XmCAccelerator**	**String**	NULL	CSG
XmNacceleratorText	**XmCAcceleratorText**	**XmString**	NULL	CSG
XmNalignment	**XmCAlignment**	**unsigned char**	dynamic	CSG
XmNfontList	**XmCFontList**	**XmFontList**	dynamic	CSG
XmNlabelInsensitive-Pixmap	**XmCLabelInsensitive-Pixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNlabelPixmap	**XmCLabelPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNlabelString	**XmCXmString**	**XmString**	dynamic	CSG
XmNlabelType	**XmCLabelType**	**unsigned char**	XmSTRING	CSG
XmNmarginBottom	**XmCMarginBottom**	**Dimension**	0	CSG
XmNmarginHeight	**XmCMarginHeight**	**Dimension**	2	CSG
XmNmarginLeft	**XmCMarginLeft**	**Dimension**	0	CSG
XmNmarginRight	**XmCMarginRight**	**Dimension**	0	CSG
XmNmarginTop	**XmCMarginTop**	**Dimension**	0	CSG
XmNmarginWidth	**XmCMarginWidth**	**Dimension**	2	CSG
XmNmnemonic	**XmCMnemonic**	**KeySym**	NULL	CSG
XmNmnemonicCharSet	**XmCMnemonicCharSet**	**String**	XmFONTLIST-_DEFAULT_TAG	CSG
XmNrecomputeSize	**XmCRecomputeSize**	**Boolean**	True	CSG
XmNstringDirection	**XmCStringDirection**	**XmStringDirection**	dynamic	CSG

XmPrimitive Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	dynamic	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNforeground	**XmCForeground**	**Pixel**	dynamic	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	NULL	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	dynamic	CSG
XmNhighlightOnEnter	**XmCHighlightOnEnter**	**Boolean**	False	CSG
XmNhighlightPixmap	**XmCHighlightPixmap**	**Pixmap**	dynamic	CSG
XmNhighlightThickness	**XmCHighlightThickness**	**Dimension**	0	CSG
XmNnavigationType	**XmCNavigationType**	**XmNavigationType**	XmNONE	CSG
XmNshadowThickness	**XmCShadowThickness**	**Dimension**	0	CSG
XmNtopShadowColor	**XmCTopShadowColor**	**Pixel**	dynamic	CSG
XmNtopShadowPixmap	**XmCTopShadowPixmap**	**Pixmap**	dynamic	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	False	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	dynamic	CSG
XmNuserData	**XmCUserData**	**XtPointer**	NULL	CSG

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	dynamic	CSG
XmNancestorSensitive	**XmCSensitive**	**Boolean**	dynamic	G
XmNbackground	**XmCBackground**	**Pixel**	dynamic	CSG
XmNbackgroundPixmap	**XmCPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	XtDefaultForeground	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	dynamic	CG
XmNdepth	**XmCDepth**	**int**	dynamic	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	NULL	C
XmNheight	**XmCHeight**	**Dimension**	dynamic	CSG
XmNinitialResources-Persistent	**XmCInitialResources-Persistent**	**Boolean**	True	C
XmNmappedWhen-Managed	**XmCMappedWhen-Managed**	**Boolean**	True	CSG
XmNscreen	**XmCScreen**	**Screen** *	dynamic	CG
XmNsensitive	**XmCSensitive**	**Boolean**	True	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	dynamic	CSG
XmNwidth	**XmCWidth**	**Dimension**	dynamic	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Callback Information

A pointer to the following structure is passed to each DtMenuButton callback:

```
typedef struct {
        int    reason;
        XEvent *event;
} XmAnyCallbackStruct;
```

The *reason* argument indicates why the callback was invoked; it is always DtCR_CASCADING when the **DtNcascadingCallback** is issued.

The *event* argument points to the **XEvent** that triggered the callback or NULL if the callback was not triggered by an **XEvent**.

SEE ALSO

DtCreateMenuButton( ); *Core*, *XmLabel*, *XmPrimitive*, *XmRowColumn*, in the X/Open CAE Specification, **Motif Toolkit API**; *XtGetValues*( ), *XtSetValues*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

NAME

DtSpinBox — the SpinBox widget class

SYNOPSIS

```
#include <Dt/SpinBox.h>
```

DESCRIPTION

The DtSpinBox widget is a user interface control to increment and decrement an arbitrary TextField. For example, it can be used to cycle through the months of the year or days of the month.

Widget subclassing is not supported for the DtSpinBox widget class.

Classes

The DtSpinBox widget inherits behaviour and resources from the *Core*, *Composite* and *XmManager* classes.

The class pointer is **dtSpinBoxWidgetClass**.

The class name is *DtSpinBoxWidget*.

New Resources

The following table defines a set of widget resources used by the application to specify data. The application can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or by class in a **.Xdefaults** file, the application must remove the **DtN** or **DtC** prefix and use the remaining letters. To specify one of the defined values for a resource in a **.Xdefaults** file, the application must remove the **Dt** prefix and use the remaining letters (in either lower case or upper case, but including any underscores between words). The codes in the access column indicate if the given resource can be set at creation time (C), set by using *XtSetValues*( ) (S), retrieved by using *XtGetValues*( ) (G), or is not applicable (N/A).

DtSpinBox Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
DtNarrowLayout	**DtCArrowLayout**	**unsigned char**	DtARROWS_END	CSG
DtNarrowSensitivity	**DtCArrowSensitivity**	**unsigned char**	DtARROWS-_SENSITIVE	CSG
DtNdecimalPoints	**DtCDecimalPoints**	**short**	0	CSG
DtNincrementValue	**DtCIncrementValue**	**int**	1	CSG
DtNinitialDelay	**DtCInitialDelay**	**unsigned int**	250	CSG
DtNmaximumValue	**DtCMaximumValue**	**int**	10	CSG
DtNminimumValue	**DtCMinimumValue**	**int**	0	CSG
DtNmodifyVerifyCallback	**DtCCallback**	**XtCallbackList**	NULL	C
DtNnumValues	**DtCNumValues**	**int**	0	CSG
DtNposition	**DtCPosition**	**int**	0	CSG
DtNrepeatDelay	**DtCRepeatDelay**	**unsigned int**	200	CSG
DtNspinBoxChildType	**DtCSpinBoxChildType**	**unsigned char**	XmSTRING	CG
DtNvalueChangedCallback	**DtCCallback**	**XtCallbackList**	NULL	C
DtNvalues	**DtCValues**	**XmStringTable**	NULL	CSG

DtNarrowLayout

Specifies the style and position of the SpinBox arrows. The following values are supported:

DtARROWS_FLAT_BEGINNING

The arrows are placed side by side to the right of the TextField.

DtARROWS_FLAT_END
The arrows are placed side by side to the left of the TextField.

DtARROWS_SPLIT
The down arrow is on the left and the up arrow is on the right of the TextField.

DtARROWS_BEGINNING
The arrows are stacked and placed on the left of the TextField.

DtARROWS_END
The arrows are stacked and placed on the right of the TextField.

DtNarrowSensitivity
Specifies the sensitivity of the arrows in the DtSpinBox. The following values are supported:

DtARROWS_SENSITIVE
Both arrows are active to user selection.

DtARROWS_DECREMENT_SENSITIVE
The down arrow is active and the up arrow is inactive to user selection.

DtARROWS_INCREMENT_SENSITIVE
The up arrow is active and the down arrow is inactive to user selection.

DtARROWS_INSENSITIVE
Both arrows are inactive to user selection.

DtNdecimalPoints
Specifies the position of the radix character within the numeric value when **DtNspinBoxChildType** is DtNUMERIC. This resource is used to allow for floating point values in the DtSpinBox widget.

DtNincrementValue
Specifies the amount to increment or decrement the **DtNposition** when the **DtNspinBoxChildType** is DtNUMERIC. When the Up action is activated, the **DtNincrementValue** is added to the **DtNposition** value; when the Down action is activated, the **DtNincrementValue** is subtracted from the **DtNposition** value. When **DtNspinBoxChildType** is DtSTRING, this resource is ignored.

DtNinitialDelay
Specifies the amount of time in milliseconds before the Arrow buttons will begin to spin continuously.

DtNnumValues
Specifies the number of items in the **DtNvalues** list when the **DtNspinBoxChildType** resource is DtSTRING. The value of this resource must be a positive integer. The **DtNnumValues** is maintained by the DtSpinBox widget when items are added or deleted from the **DtNvalues** list. When **DtNspinBoxChildType** is not DtSTRING, this resource is ignored.

DtNvalues
Supplies the list of strings to cycle through when the **DtNspinButtonChildType** resource is DtSTRING. When **DtNspinBoxChildType** is not DtSTRING, this resource is ignored.

DtNmaximumValue
Specifies the upper bound on the DtSpinBox's range when **DtNspinBoxChildType** is DtNUMERIC.

DtNminimumValue
Specifies the lower bound on the DtSpinBox's range when **DtNspinBoxChildType** is DtNUMERIC.

DtNmodifyVerifyCallback
Specifies the callback to be invoked just before the DtSpinBox position changes. The application can use this callback to implement new application-related logic (including setting new position spinning to, or canceling the impending action). For example, this callback can be used to stop the spinning just before wrapping at the upper and lower position boundaries. If the application sets the *doit* member of the *DtSpinBoxCallbackStruct* to False, nothing happens. Otherwise, the position changes. Reasons sent by the callback are DtCR_SPIN_NEXT, or DtCR_SPIN_PRIOR.

DtNposition
The **DtNposition** resource has a different value based on the **DtNspinBoxChildType** resource. When **DtNspinBoxChildType** is DtSTRING, the **DtNposition** is the index into the **DtNvalues** list for the current item. When the **DtNspinBoxChildType** resource is DtNUMERIC, the **DtNposition** is the integer value of the DtSpinBox that falls within the range of **DtNmaximumValue** and **DtNminimumValue**.

DtNrepeatDelay
Specifies the number of milliseconds between repeated calls to the **DtNvalueChangedCallback** while the user is spinning the DtSpinBox.

DtNspinBoxChildType
Specifies the style of the DtSpinBox. The following values are supported:

DtSTRING
The child is a string value that is specified through the **DtNvalues** resource and incremented and decremented by changing the **DtNposition** resource.

DtNUMERIC
The child is a numeric value that is specified through the **DtNposition** resource and incremented according to the **DtNincrementValue** resource.

DtNvalueChangedCallback
Specifies the callback to be invoked whenever the value of the **DtNposition** resource is changed through the use of the spinner arrows. The **DtNvalueChangedCallback** passes the *DtSpinBoxCallbackStruct call_data* structure.

Inherited Resources

The DtSpinBox widget inherits behaviour and resources from the following named superclasses. For a complete description of each resource, see the entry in X/Open CAE Specification, **Motif Toolkit API** for that superclass.

XmManager Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNbottom-ShadowColor	**XmCBottom-ShadowColor**	**Pixel**	dynamic	CSG
XmNbottom-ShadowPixmap	**XmCBottom-ShadowPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNforeground	**XmCForeground**	**Pixel**	dynamic	CSG
XmNhelpCallback	**XmCCallback**	**XtCallbackList**	NULL	C
XmNhighlightColor	**XmCHighlightColor**	**Pixel**	dynamic	CSG
XmNhighlightPixmap	**XmCHighlightPixmap**	**Pixmap**	dynamic	CSG
XmNinitialFocus	**XmCInitialFocus**	**Widget**	NULL	CSG
XmNnavigationType	**XmCNavigationType**	**XmNavigationType**	dynamic	CSG
XmNshadowThickness	**XmCShadowThickness**	**Dimension**	dynamic	CSG
XmNstringDirection	**XmCStringDirection**	**XmStringDirection**	dynamic	CG
XmNtopShadowColor	**XmCTopShadowColor**	**Pixel**	dynamic	CSG
XmNtopShadowPixmap	**XmCTopShadowPixmap**	**Pixmap**	dynamic	CSG
XmNtraversalOn	**XmCTraversalOn**	**Boolean**	dynamic	CSG
XmNunitType	**XmCUnitType**	**unsigned char**	dynamic	CSG
XmNuserData	**XmCUserData**	**XtPointer**	NULL	CSG

Composite Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNchildren	**XmCReadOnly**	**WidgetList**	NULL	G
XmNinsertPosition	**XmCInsertPosition**	**XtOrderProc**	default procedure	CSG
XmNnumChildren	**XmCReadOnly**	**Cardinal**	0	G

Core Resource Set				
Name	**Class**	**Type**	**Default**	**Access**
XmNaccelerators	**XmCAccelerators**	**XtAccelerators**	dynamic	CSG
XmNancestorSensitive	**XmCSensitive**	**Boolean**	dynamic	G
XmNbackground	**XmCBackground**	**Pixel**	dynamic	CSG
XmNbackgroundPixmap	**XmCPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNborderColor	**XmCBorderColor**	**Pixel**	XtDefaultForeground	CSG
XmNborderPixmap	**XmCPixmap**	**Pixmap**	XmUNSPECIFIED-_PIXMAP	CSG
XmNborderWidth	**XmCBorderWidth**	**Dimension**	0	CSG
XmNcolormap	**XmCColormap**	**Colormap**	dynamic	CG
XmNdepth	**XmCDepth**	**int**	dynamic	CG
XmNdestroyCallback	**XmCCallback**	**XtCallbackList**	NULL	C
XmNheight	**XmCHeight**	**Dimension**	dynamic	CSG
XmNinitialResources-Persistent	**XmCInitialResources-Persistent**	**Boolean**	True	C
XmNmappedWhen-Managed	**XmCMappedWhen-Managed**	**Boolean**	True	CSG
XmNscreen	**XmCScreen**	**Screen** *	dynamic	CG
XmNsensitive	**XmCSensitive**	**Boolean**	True	CSG
XmNtranslations	**XmCTranslations**	**XtTranslations**	dynamic	CSG
XmNwidth	**XmCWidth**	**Dimension**	dynamic	CSG
XmNx	**XmCPosition**	**Position**	0	CSG
XmNy	**XmCPosition**	**Position**	0	CSG

Callback Information

A pointer to the following structure is passed to each DtSpinBox callback:

```
typedef struct {
        int             reason;
        XEvent          *event;
        Widget          widget;
        Boolean         doit;
        int             position;
        XmString        value;
        Boolean         crossed_boundary;
} DtSpinBoxCallbackStruct;
```

The *reason* argument indicates why the callback was invoked. There are three possible reasons for this callback to be issued. The reason is DtCR_OK when this is the first call to the callback at the beginning of a spin or if it is a single activation of the spin arrows. If the DtSpinBox is in the process of being continuously spun, then the reason will be DtCR_SPIN_NEXT or DtCR_SPIN_PRIOR, depending on the arrow that is spinning.

The *event* argument points to the **XEvent** that triggered the callback. It can be NULL when the DtSpinBox is continuously spinning.

The *widget* argument is the widget identifier for the text widget that has been affected by the spin.

The *doit* argument is set only when the *call_data* comes from the **DtNmodifyVerifyCallback**. It indicates that the action that caused the callback to be called should be performed. The action is not performed if *doit* is set to False.

The *position* argument is the new value of the **DtNposition** resource as a result of the spin.

The *value* argument is the new **XmString** value displayed in the Text widget as a result of the spin. The application must copy this string if it is used beyond the scope of the *call_data* structure.

The *crossed_boundary* argument is True when the spinbox cycles. This is the case when a **DtNspinBoxChildType** of DtSTRING wraps from the first item to the last or the last item to the first. In the case of the **DtNspinBoxChildType** of DtNUMERIC, the boundary is crossed when the DtSpinBox cycles from the maximum value to the minimum or vice versa.

SEE ALSO

DtCreateSpinButton( ), *DtSpinButtonAddItem*( ), *DtSpinButtonDeletePos*( ), *DtSpinButtonSetItem*( ); *Composite*, *Core*, *XmManager*, *XmText*, *XmTextField*, in the X / Open CAE Specification, **Motif Toolkit API**; *XtGetValues*( ), *XtSetValues*( ) in the X / Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

4.8 XCDE Widget Convenience Functions

This section defines the functions, macros and external variables that provide XCDE convenience functions for the XCDE-specific widgets to support application portability at the C-language source level.

NAME

DtComboBoxAddItem — add an item to the ComboBox widget

SYNOPSIS

```
#include <Dt/ComboBox.h>

void DtComboBoxAddItem(Widget w,
                       XmString item,
                       int pos,
                       Boolean unique);
```

DESCRIPTION

The *DtComboBoxAddItem*( ) function adds the given item to the DtComboBox at the given position.

The *w* argument specifies the DtComboBox widget ID.

The *item* argument specifies the **XmString** for the new item.

The *pos* argument specifies the position of the new item.

The *unique* argument specifies if this item should duplicate an identical item or not.

RETURN VALUE

The *DtComboBoxAddItem*( ) function returns no value.

APPLICATION USAGE

The functions *DtComboBoxAddItem*( ) and *DtComboBoxDeletePos*( ) have different naming conventions (Item versus Pos) because of the objects they are manipulating. The Item is a string to be added, the Pos is a numeric position number.

SEE ALSO

DtComboBox, *DtComboBoxDelPos*( ), *DtComboBoxSetItem*( ), *DtComboBoxSelectItem*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtComboBoxDeletePos — delete a DtComboBox item

SYNOPSIS

```
#include <Dt/ComboBox.h>

void DtComboBoxDeletePos(Widget w,
                         int pos);
```

DESCRIPTION

The *DtComboBoxDeletePos*( ) function deletes a specified item from a DtComboBox widget.

The *w* argument specifies the DtComboBox widget ID.

The *pos* argument specifies the position of the item to be deleted.

RETURN VALUE

The *DtComboBoxDeletePos*( ) function returns no value.

APPLICATION USAGE

The functions *DtComboBoxAddItem*( ) and *DtComboBoxDeletePos*( ) have different naming conventions (Item versus Pos) because of the objects they are manipulating. The Item is a string to be added, the Pos is a numeric position number.

SEE ALSO

DtComboBox, *DtComboBoxAddItem*( ), *DtComboBoxSetItem*( ), *DtComboBoxSelectItem*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtComboBoxSelectItem — select a DtComboBox item

SYNOPSIS

```
#include <Dt/ComboBox.h>

void DtComboBoxSelectItem(Widget w,
                          XmString item);
```

DESCRIPTION

The *DtComboBoxSelectItem*( ) function selects an item in the XmList of the DtComboBox widget.

The *w* argument specifies the DtComboBox widget ID.

The *item* argument specifies the **XmString** of the item to be selected. If the *item* is not found on the list, *DtComboBoxSelectItem*( ) notifies the user via the *XtWarning*( ) function.

RETURN VALUE

The *DtComboBoxSelectItem*( ) function returns no value.

SEE ALSO

DtComboBox, *DtComboBoxAddItem*( ), *DtComboBoxDeletePos*( ), *DtComboBoxSetItem*( ); *XtWarning*( ) in the X / Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

NAME

DtComboBoxSetItem — set an item in the DtComboBox list

SYNOPSIS

```
#include <Dt/ComboBox.h>

void DtComboBoxSetItem(Widget w,
                       XmString item);
```

DESCRIPTION

The *DtComboBoxSetItem*( ) function selects an item in the XmList of the given DtComboBox widget and makes it the first visible item in the list.

The *w* argument specifies the DtComboBox widget ID.

The *item* argument specifies the **XmString** for the item to be set in the DtComboBox. If the *item* is not found on the list, *DtComboBoxSetItem*( ) notifies the user via the *XtWarning*( ) function.

RETURN VALUE

The *DtComboBoxSetItem*( ) function returns no value.

SEE ALSO

DtComboBox, *DtComboBoxAddItem*( ), *DtComboBoxDeletePos*( ), *DtComboBoxSelectItem*( ); *XtWarning*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

NAME

DtCreateComboBox — the ComboBox widget creation function

SYNOPSIS

```
#include <Dt/ComboBox.h>

Widget DtCreateComboBox(Widget parent,
                        String name,
                        ArgList arglist,
                        Cardinal argcount);
```

DESCRIPTION

The *DtCreateComboBox*( ) function creates an instance of a ComboBox widget and returns the associated widget ID.

The *parent* argument specifies the parent widget ID.

The *name* argument specifies the name of the created widget.

The *arglist* argument specifies the argument list.

The *argcount* argument specifies the number of attribute/value pairs in the argument list.

RETURN VALUE

Upon successful completion, the *DtCreateComboBox*( ) function returns the ComboBox widget ID.

SEE ALSO

DtComboBox.

CHANGE HISTORY

First released in Issue 1.

NAME

DtCreateMenuButton — the MenuButton widget creation function

SYNOPSIS

```
#include <Dt/MenuButton.h>

Widget DtCreateMenuButton(Widget parent,
                          String name,
                          ArgList arglist,
                          Cardinal argcount);
```

DESCRIPTION

The *DtCreateMenuButton*( ) function creates an instance of a MenuButton widget and returns the associated widget ID.

The *parent* argument specifies the parent widget ID.

The *name* argument specifies the name of the created widget.

The *arglist* argument specifies the argument list.

The *argcount* argument specifies the number of attribute/value pairs in the argument list.

RETURN VALUE

Upon successful completion, the *DtCreateMenuButton*( ) function returns the MenuButton widget ID.

SEE ALSO

DtMenuButton.

CHANGE HISTORY

First released in Issue 1.

NAME

DtCreateSpinBox — the SpinBox widget creation function

SYNOPSIS

```
#include <Dt/SpinBox.h>

Widget DtCreateSpinBox(Widget parent,
                       String name,
                       ArgList arglist,
                       Cardinal argcount);
```

DESCRIPTION

The *DtCreateSpinBox*( ) function creates an instance of a SpinBox widget and returns the associated widget ID.

The *parent* argument specifies the parent widget ID.

The *name* argument specifies the name of the created widget.

The *arglist* argument specifies the argument list.

The *argcount* argument specifies the number of attribute/value pairs in the argument list.

RETURN VALUE

Upon successful completion, the *DtCreateSpinBox*( ) function returns the SpinBox widget ID.

SEE ALSO

DtSpinBox.

CHANGE HISTORY

First released in Issue 1.

NAME

DtSpinBoxAddItem — add an item to the DtSpinBox

SYNOPSIS

```
#include <Dt/SpinBox.h>

void DtSpinBoxAddItem(Widget w,
                      XmString item,
                      int pos);
```

DESCRIPTION

The *DtSpinBoxAddItem*( ) function adds the given item to the DtSpinBox at the given position.

The *w* argument specifies the widget ID.

The *item* argument specifies the **XmString** for the new item.

The *pos* argument specifies the position of the new item.

RETURN VALUE

The *DtSpinBoxAddItem*( ) function returns no value.

SEE ALSO

DtSpinBox, *DtSpinBoxDeletePos*( ), *DtSpinBoxSetItem*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtSpinBoxDeletePos — delete a DtSpinBox item

SYNOPSIS

```
#include <Dt/SpinBox.h>

void DtSpinBoxDeletePos(Widget w,
                        int pos);
```

DESCRIPTION

The *DtSpinBoxDeletePos*( ) function deletes a specified item from a DtSpinBox widget.

The *w* argument specifies the widget ID.

The *pos* argument specifies the position of the item to be deleted. A value of 1 means the first item in the list; zero means the last item.

RETURN VALUE

The *DtSpinBoxDeletePos*( ) function returns no value.

SEE ALSO

DtSpinBox, *DtSpinBoxAddItem*( ), *DtSpinBoxSetItem*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtSpinBoxSetItem — set an item in the DtSpinBox list

SYNOPSIS

```
#include <Dt/SpinBox.h>

void DtSpinBoxSetItem(Widget w,
                      XmString item);
```

DESCRIPTION

The *DtSpinBoxSetItem*( ) function selects an item in the list of the given DtSpinBox widget and makes it the current value.

The *w* argument specifies the widget ID.

The *item* argument specifies the **XmString** for the item to be set in the DtSpinBox. If the *item* is not found on the list, *DtSpinBoxSetItem*( ) notifies the user via the *XtWarning*( ) function.

RETURN VALUE

The *DtSpinBoxSetItem*( ) function returns no value.

SEE ALSO

DtSpinBox, *DtSpinBoxAddItem*( ), *DtSpinBoxDeletePos*( ); *XtWarning*( ) in the X/Open CAE Specification, **Window Management**: **X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

4.9 XCDE Widget Headers

This section describes the contents of headers used by the XCDE drag and drop functions, macros and external variables.

Headers contain the definition of symbolic constants, common structures, preprocessor macros and defined types. Each function in Section 4.8 specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

NAME

Dt/ComboBox.h — DtComboBox widget definitions

SYNOPSIS

```
#include <Dt/ComboBox.h>
```

DESCRIPTION

The **<Dt/ComboBox.h>** header defines the following structure:

```
typedef struct {
        int reason;
        XEvent *event;
        XmString item_or_text;
        int item_position;
} DtComboBoxCallbackStruct;
```

The header declares the following constants:

```
DtALIGNMENT_BEGINNING
DtALIGNMENT_CENTER
DtALIGNMENT_END
DtCR_SELECT
DtDROP_DOWN_COMBO_BOX
DtDROP_DOWN_LIST
```

The header declares the following as functions:

```
void DtComboBoxAddItem(Widget w,
                       XmString item,
                       int pos,
                       Boolean unique);

void DtComboBoxDeletePos(Widget w,
                         int pos);

void DtComboBoxSelectItem(Widget w,
                          XmString item);

void DtComboBoxSetItem(Widget w,
                       XmString item);
```

CHANGE HISTORY

First released in Issue 1.

NAME

Dt/MenuButton.h — DtMenuButton widget definitions

SYNOPSIS

```
#include <Dt/MenuButton.h>
```

DESCRIPTION

The **<Dt/MenuButton.h>** header defines the following constant:

```
DtCR_CASCADING
```

The header declares the following as a function:

```
Widget DtCreateMenuButton(Widget parent,
                          String name,
                          ArgList arglist,
                          Cardinal argcount);
```

CHANGE HISTORY

First released in Issue 1.

NAME

Dt/SpinBox.h — DtSpinBox widget definitions

SYNOPSIS

```
#include <Dt/SpinBox.h>
```

DESCRIPTION

The **<Dt/SpinBox.h>** header defines the following structure:

```
typedef struct {
        int reason;
        XEvent *event;
        Widget widget;
        Boolean doit;
        int position;
        XmString value;
        Boolean crossed_boundary;
} DtSpinBoxCallbackStruct;
```

The header declares the following constants:

```
DtARROWS_FLAT_BEGINNING
DtARROWS_FLAT_END
DtARROWS_SPLIT
DtARROWS_BEGINNING
DtARROWS_END
DtARROWS_SENSITIVE
DtARROWS_DECREMENT_SENSITIVE
DtARROWS_INCREMENT_SENSITIVE
DtARROWS_INSENSITIVE
DtNUMERIC
DtALIGNMENT_BEGINNING
DtALIGNMENT_CENTER
DtALIGNMENT_END
DtCR_SPIN_NEXT
DtCR_SPIN_PRIOR
```

The header declares the following as functions:

```
void DtSpinBoxAddItem(Widget w,
                      XmString item,
                      int pos);

void DtSpinBoxDeletePos(Widget w,
                        int pos);

void DtSpinBoxSetItem(Widget w,
                      XmString item);
```

CHANGE HISTORY

First released in Issue 1.

Chapter 5

Miscellaneous Desktop Services

5.1 Introduction

This section defines the functions, macros and external variables that provide miscellaneous XCDE services to support application portability at the C-language source level.

5.2 Functions

The following functions initialise the Desktop Services library.

NAME

DtInitialize, DtAppInitialize — initialise the Desktop Services library

SYNOPSIS

```
#include <Dt/Dt.h>

Boolean DtInitialize(Display *display,
                     Widget widget,
                     char *name,
                     char *tool_class)

Boolean DtAppInitialize(XtAppContext app_context,
                        Display *display,
                        Widget widget,
                        char *name,
                        char *tool_class)
```

DESCRIPTION

These functions perform the one-time initialisation in the Desktop Services library. Applications must call either *DtInitialize*( ) or *DtAppInitialize*( ) before calling any other Desktop Services library routines.

The difference between these two functions is whether *app_context* is specified. *DtInitialize*( ) uses the default Intrinsic *XtAppContext*.

The *app_context* argument is the application context, *display* is the X display connection, *widget* is the application's top-level Widget, *name* is the application name and *tool_class* is the application class.

RETURN VALUES

Upon successful completion, *DtAppInitialize*( ) and *DtInitialize*( ) return True if the library has been correctly initialised; otherwise, they return False.

SEE ALSO

<Dt/Dt.h>; *XtAppInitialize*( ), *XtToolkitInitialize*( ), *XtCreateApplicationContext*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

5.3 Headers

This section describes the contents of the header used by miscellaneous XCDE message service functions, macros and external variables.

Headers contain the definition of symbolic constants, common structures, preprocessor macros and defined types. Each function in Section 5.2 specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

NAME

Dt/Dt.h — miscellaneous desktop definitions

SYNOPSIS

```
#include <Dt/Dt.h>
```

DESCRIPTION

The **<Dt/Dt.h>** header contains miscellaneous public constant and function declaration for the XCDE library.

The header defines several constants that can be used to determine the version of the library used to compile an application and the version of the library with which an application is currently linked.

The header defines the following constants that represent the library compile-time version:

DtVERSION	An integer specifying the major version number
DtREVISION	An integer specifying the minor version number
DtUPDATE_LEVEL	An integer specifying the patch release level
DtVERSION_NUMBER	An integer combining the major, minor and patch release numbers. It is derived from the following formula: `(10000 * DtVERSION + 100 * DtRevision + DtUPDATE_LEVEL)`
DtVERSION_STRING	A string containing a description of the version and the version number

The header defines the following constants that represent the library run-time version:

```
extern int DtVersion
extern char *DtVersionString
```

DtVersion is an integer equivalent to DtVERSION_NUMBER at the time the library was created. **DtVersionString** is a string equivalent to DtVERSION_STRING at the time the library was created.

The header declares the following as functions:

```
Boolean DtInitialize(Display *display,
                     Widget widget,
                     char *name,
                     char *tool_class)

Boolean DtAppInitialize(XtAppContext app_context,
                        Display *display,
                        Widget widget,
                        char *name,
                        char *tool_class)
```

CHANGE HISTORY

First released in Issue 1.

Chapter 6

Message Services

6.1 Introduction

The XCDE messaging service provides APIs and supporting components for passing multicast and point-to-point messages between desktop applications, both across networks and within hosts. Message protocols, defined in Section 6.6 on page 363, use the messaging service to achieve control and data integration between desktop applications.

The messaging service is based on pattern matching: applications wishing to receive messages register patterns that describe the desired messages; applications sending messages format the message as a description of the service being requested or the event being announced. The messaging service then routes the messages to the interested application. Since applications need not directly address each other, this provides the ability to restructure applications and swap in different implementations without modifying other applications.

The messaging service supports messaging both within a single user's session (session-scoped messaging) and between users (file-scoped messaging). The messaging service explicitly supports messages that ask for services (requests, and the associated replies), and messages that announce events (notices).

The messaging service permits applications to register patterns at installation time; by consulting these patterns, the messaging service can determine that an application not currently running is interested in the message, and start the application.

The XCDE messaging service is based on the ToolTalk facilities from Sun Microsystems, Inc.; this document generally refers to the messaging service as ''the ToolTalk service.''

6.2 Functions

This section defines the functions, macros and external variables that provide XCDE message services to support application portability at the C-language source level.

NAME

tt_X_session — return the session associated with an X window system display

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_X_session(const char *xdisplaystring);
```

DESCRIPTION

The *tt_X_session*( ) function returns the session associated with the named X window system display.

The application can call *tt_X_session*( ) before it calls *tt_open*( ).

The *xdisplaystring* argument is the name of an X display server; for example, **somehost:0** or **:0**.

RETURN VALUE

Upon successful completion, the *tt_X_session*( ) function returns the identifier for the ToolTalk session associated with the named X window system display. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_SESSION
: The *xdisplaystring* does not name an X display.

TT_ERR_POINTER
: The *xdisplaystring* is NULL.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_open*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_bcontext_join — add a byte-array value to the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_bcontext_join(const char *slotname,
                           const unsigned char *value,
                           int length);
```

DESCRIPTION

The *tt_bcontext_join*( ) function adds the given byte-array value to the list of values for the named contexts of all patterns. The context is compared to currently registered patterns for the procid. If a pattern has a slot with the specified name, the given byte-array value is added to the list of values for that slot.

The *slotname* argument is the name of the context. The *value* argument is the value to be added. The *length* argument is the length in bytes of the value.

RETURN VALUE

Upon successful completion, the *tt_bcontext_join*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_bcontext_quit — remove a byte-array value from the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_bcontext_quit(const char *slotname,
                           const unsigned char *value,
                           int length);
```

DESCRIPTION

The *tt_bcontext_quit*( ) function removes the given byte-array value from the list of values for the contexts of all patterns. The context is compared to currently registered patterns for the procid. If a pattern has a slot with the specified name, the given byte string value is removed from the list of values for that slot. If there are duplicate values, only one value is removed.

The *slotname* argument is the name of the context. The *value* argument is the value to be removed. The *length* argument is the length in bytes of the value.

RETURN VALUE

Upon successful completion, the *tt_bcontext_quit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_close — close the current default procid

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_close(void);
```

DESCRIPTION

The *tt_close*( ) function closes the current default procid.

RETURN VALUE

Upon successful completion, the *tt_close*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

APPLICATION USAGE

When the *tt_close*( ) function call is successful, the procid will no longer be active. For any subsequent API calls the process must, therefore, first call *tt_default_procid_set*( ) to specify a procid.

SEE ALSO

<Tt/tt_c.h>, *tt_open*( ), *tt_context_join*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_context_join — add a string value to the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_context_join(const char *slotname,
                          const char *value);
```

DESCRIPTION

The *tt_context_join*( ) function adds the given string value to the list of values for the context of all patterns.

The context is compared to currently registered patterns for the procid. If a pattern has a slot with the specified name, the given string value is added to the list of values for that slot.

The *slotname* argument is the name of the context. The *value* argument is the value to be added.

RETURN VALUE

Upon successful completion, the *tt_context_join*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_context_quit — remove a string value from the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_context_quit(const char *slotname,
                          const char *value);
```

DESCRIPTION

The *tt_context_quit*( ) function removes the given string value from the list of values for the contexts of all patterns.

The context is compared to currently registered patterns for the procid. If a pattern has a slot with the specified name, *tt_context_quit*( ) removes the given string value from the list of values for that slot. If there are duplicate values, only one value is removed.

The *slotname* argument is the name of the context. The *value* argument is the value to be added.

RETURN VALUE

Upon successful completion, the *tt_context_quit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_default_file — return the current default file

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_default_file(void);
```

DESCRIPTION

The *tt_default_file*( ) function returns the current default file.

When the application joins a file, the file becomes the default.

RETURN VALUE

Upon successful completion, the *tt_default_file*( ) function returns the pointer to a character string that specifies the current default file. If the pointer is NULL, no default file is set. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_file_join*( ), *tt_default_file_set*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_default_file_set — set the default file to a file

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_default_file_set(const char *docid);
```

DESCRIPTION

The *tt_default_file_set*( ) function sets the default file to the specified file.

The *docid* argument is a pointer to a character string that specifies the file that is to be the default file.

RETURN VALUE

Upon successful completion, the *tt_default_file_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

TT_ERR_FILE
: The specified file does not exist or it is inaccessible.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_default_procid — identify the current default process

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_default_procid(void);
```

DESCRIPTION

The *tt_default_procid*( ) function retrieves the current default procid for the process.

RETURN VALUE

Upon successful completion, the *tt_default_procid*( ) function returns the pointer to a character string that uniquely identifies the current default process. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_default_procid_set — set the current default procid

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_default_procid_set(const char *procid);
```

DESCRIPTION

The *tt_default_procid_set*( ) function sets the current default procid.

The *procid* argument is the name of process that is to be the default process.

RETURN VALUE

Upon successful completion, the *tt_default_procid_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>, *tt_open*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_default_ptype — retrieve the current default ptype

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_default_ptype(void);
```

DESCRIPTION

The *tt_default_ptype*( ) function retrieves the current default ptype.

When the application declares a ptype, the ptype becomes the default.

RETURN VALUE

Upon successful completion, the *tt_default_ptype*( ) function returns a pointer to a character string that uniquely identifies the current default process type. If the pointer is NULL, no default ptype is set. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptype_declare*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_default_ptype_set — set the default ptype

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_default_ptype_set(const char *ptid);
```

DESCRIPTION

The *tt_default_ptype_set*( ) function sets the default ptype.

The *ptid* argument must be the character string that uniquely identifies the process that is to be the default process.

RETURN VALUE

Upon successful completion, the *tt_default_ptype_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_default_session — retrieve the current default session identifier

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_default_session(void);
```

DESCRIPTION

The *tt_default_session*( ) function retrieves the current default session identifier.

RETURN VALUE

Upon successful completion, the *tt_default_session*( ) function returns the pointer to the unique identifier for the current default session. If the pointer is NULL, no default session is set. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

APPLICATION USAGE

A session can have more than one session identifier. This means that the application cannot compare the result of *tt_default_session*( ) with the result of *tt_message_session*( ) to verify that the message was sent in the default session.

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_default_session_set — set the current default session identifier

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_default_session_set(const char *sessid);
```

DESCRIPTION

The *tt_default_session_set*( ) function sets the current default session identifier.

The ToolTalk service uses the initial user session as the default session and supports one session per procid. The application can make this call before it calls *tt_open*( ) to specify the session to which it wants to connect.

The *sessid* argument is a pointer to the unique identifier for the session in which the procid is interested.

RETURN VALUE

Upon successful completion, the *tt_default_session_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

APPLICATION USAGE

To change to another opened session, the application must use the *tt_default_procid_set*( ) function.

To join other sessions, the procid must first set the new session as the default session, and then initialise and register with the ToolTalk service. The calls required must be in the following order:

tt_default_session_set ( )
tt_open ( )

The *tt_open*( ) may create another ToolTalk procid, the connection to which is identified by a procid. Only one ToolTalk session per procid is allowed. (However, multiple procids are allowed in a client.) There are no API calls to determine to which session a particular procid is connected. If it is important for the application to know the session to which it is connected, it must make the following calls in the indicated order:

tt_open ( )
tt_default_session ( )

The application can then store the information by indexing it by the procid returned by the *tt_open*( ) call.

SEE ALSO

<Tt/tt_c.h>, *tt_open*( ), *tt_default_procid*( ), *tt_default_session*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_error_int — return an integer error object that encodes the code

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_error_int(Tt_status ttrc);
```

DESCRIPTION

The *tt_error_int*( ) function returns an integer error object that encodes a **Tt_status** return value.

The *ttrc* argument is the **Tt_status** code that is to be encoded.

RETURN VALUE

Upon successful completion, the *tt_error_int*( ) function returns the encoded **Tt_status** code.

APPLICATION USAGE

The integer error objects are negative integers; an application should use this call only when the valid integer values are non-negative.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_error_pointer — return a pointer to an error object that encodes the code

SYNOPSIS

```
#include <Tt/tt_c.h>

void *tt_error_pointer(Tt_status ttrc);
```

DESCRIPTION

The *tt_error_pointer*( ) function returns a pointer to an error object that encodes a **Tt_status** return value.

The *ttrc* argument is the **Tt_status** code that is to be encoded.

RETURN VALUE

Upon successful completion, the *tt_error_pointer*( ) function returns a pointer to the encoded **Tt_status** code.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_fd — return a file descriptor

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_fd(void);
```

DESCRIPTION

The *tt_fd*( ) function returns a file descriptor. The returned file descriptor alerts the process that a message has arrived for the default procid in the default session.

File descriptors are either active or inactive. When the file descriptor becomes active, the process must call *tt_message_receive*( ) to receive the message.

RETURN VALUE

Upon successful completion, the *tt_fd*( ) function returns the file descriptor for the current procid. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The current default process identifier is out of date or invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

APPLICATION USAGE

The application must have a separate file descriptor for each procid. To get an associated file descriptor, the application should use *tt_fd*( ) each time it calls *tt_open*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_open*( ), *tt_int_error*( ), *tt_message_receive*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_file_copy — copy objects from one file to a new file

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_file_copy(const char *oldfilepath,
                       const char *newfilepath);
```

DESCRIPTION

The *tt_file_copy*( ) function copies all objects that exist on the specified file to a new file. If any objects already exist on *newfilepath*, they are not overwritten by the copy (that is, they are not removed.)

The *oldfilepath* argument is a pointer to the name of the file whose objects are to be copied. The *newfilepath* argument is a pointer to the name of the file on which to create the copied objects.

RETURN VALUE

Upon successful completion, the *tt_file_copy*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_FILE
: The specified file does not exist or it is inaccessible.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PATH
: The specified pathname included an unsearchable directory.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_file_move*( ), *tt_file_destroy*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_file_destroy — remove objected rooted on a file

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_file_destroy(const char *filepath);
```

DESCRIPTION

The *tt_file_destroy*( ) function removes all objects that exist on the files and directories rooted at *filepath*. The application must call this function when the application unlinks a file or removes a directory.

The *filepath* argument is a pointer to the pathname of the file or directory to be removed.

RETURN VALUE

Upon successful completion, the *tt_file_destroy*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_ACCESS
: The user does not have the necessary access to the object and/or the process.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_FILE
: The specified file does not exist or it is inaccessible.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PATH
: The specified pathname included an unsearchable directory.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_file_copy*( ), *tt_file_move*( ); *rmdir*, *unlink*( ) in the X/Open CAE Specification, **System Interface Definitions**, **Issue 4, Version 2**.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_file_join — register interest in messages involving a file

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_file_join(const char *filepath);
```

DESCRIPTION

The *tt_file_join*( ) function informs the ToolTalk service that the process is interested in messages that involve the specified file.

The ToolTalk service adds this file value to any currently registered patterns. The named file becomes the default file.

When the process joins a file, the ToolTalk service updates the file field of its registered patterns. The *tt_file_join*( ) call causes the pattern's ToolTalk session to be stored in the database.

The *filepath* argument is a pointer to the pathname of the file in which the process is interested.

RETURN VALUE

Upon successful completion, the *tt_file_join*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PATH
: The specified pathname included an unsearchable directory.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_file_move — move objects from one file to another

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_file_move(const char *oldfilepath,
                       const char *newfilepath);
```

DESCRIPTION

The *tt_file_move*( ) function destroys all objects that exist on the files and directories rooted at *newfilepath*, then moves all objects that exist on *oldfilepath* to *newfilepath*.

If *oldfilepath* and *newfilepath* reside in the same file system, *tt_file_move*( ) replaces *oldfilepath* with *newfilepath* in the path associated with every object in that file system; that is, all the objects in the directory tree rooted at *oldfilepath* are overlaid onto *newfilepath*. In this mode, the behaviour of *tt_file_move*( ) is similar to *rename*( ).

If *oldfilepath* and *newfilepath* reside in different file systems, neither can be a directory.

The *oldfilepath* argument is the name of the file or directory whose objects are to be moved. The *newfilepath* argument is the name of the file or directory to which the objects are to be moved.

RETURN VALUE

Upon successful completion, the *tt_file_move*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_ACCESS
: The user does not have the necessary access to the object and/or the process.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_FILE
: The specified file does not exist or it is inaccessible.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PATH
: The specified pathname included an unsearchable directory, or *oldfilepath* and *newfilepath* reside in different file systems, and either is a directory.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_file_copy*( ), *tt_file_destroy*( ); *rename*( ) in the X/Open CAE Specification, **System Interface Definitions**, **Issue 4, Version 2**.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_file_netfile — map between local and canonical pathnames on the local host

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_file_netfile(const char *filename);
```

DESCRIPTION

The *tt_file_netfile*( ) function converts a local pathname to a *netfilename*, a form that can be passed to other hosts on the network and converted back to a local pathname for the same file with *tt_netfile_file*( ).

The *filename* argument is a pathname (absolute or relative) that is valid on the local host. Every component of *filename* must exist, except that the last component need not exist.

RETURN VALUE

Upon successful completion, the *tt_file_netfile*( ) function returns a freshly allocated null-terminated string of unspecified format, which can be passed to *tt_netfile_file*( ) or *tt_host_netfile_file*( ); otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_ERR_PATH
: The *filename* argument is a path that is not valid on this host.

APPLICATION USAGE

The *tt_file_netfile*( ), *tt_netfile_file*( ), *tt_host_file_netfile*( ) and *tt_host_netfile_file*( ) functions allow an application to determine a path valid on remote hosts, perhaps for purposes of constructing a command string valid for remote execution on that host. By composing the two calls, paths for files not accessible from the current host can be constructed. For example, if path **/sample/file** is valid on host A, a program running on host B can use

```
tt_host_netfile_file("C", tt_host_file_netfile("A", "/sample/file"))
```

to determine a path to the same file valid on host C, if such a path is possible.

The *netfile* string returned by *tt_file_netfile*( ) should be considered opaque; the content and format of the strings are not a public interface. These strings can be safely copied (with *strcpy*( ) or similar methods), written to files, or transmitted to other processes, perhaps on other hosts.

Allocated strings should be freed using either *tt_free*( ) or *tt_release*( ).

The *tt_open*( ) function need not be called before *tt_file_netfile*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_netfile_file*( ), *tt_host_file_netfile*( ), *tt_host_netfile_file*( ), *tt_open*( ), *tt_free*( ), *tt_release*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_file_objects_query — find all objects in the named file

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_file_objects_query(const char *filepath,
                                Tt_filter_function filter,
                                void *context,
                                void *accumulator);
```

DESCRIPTION

The *tt_file_objects_query*( ) function instructs the ToolTalk service to find all objects in the named file and pass the objids to the filter function. The context pointer and accumulator pointer initially specified are also passed to the filter function.

As the ToolTalk service finds each object, it calls the filter function, passing the objid of the object and the two application-supplied pointers. The filter function performs its computation and returns a **Tt_filter_action** value that tells the query function whether to continue or to stop. **Tt_filter_action** values are:

TT_FILTER_CONTINUE
: The query function should continue.

TT_FILTER_STOP
: The query function should stop.

The *filepath* argument is the name of the file to be searched for objects. The *filter* argument is the filter function to which the objids are to be passed. The *context* argument is a pointer to any information the filter needs to execute. The ToolTalk service does not interpret this argument, but passes it directly to the filter function. The *accumulator* argument is a pointer to where the filter is to store the results of the query and filter operations. The ToolTalk service does not interpret this argument, but passes it directly to the filter function.

RETURN VALUE

Upon successful completion, the *tt_file_objects_query*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PATH
: The specified pathname included an unsearchable directory.

TT_WRN_STOPPED
: The query operation being performed was halted by **Tt_filter_function**.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_file_quit — register lack of interest in messages that involve a file

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_file_quit(const char *filepath);
```

DESCRIPTION

The *tt_file_quit*( ) function informs the ToolTalk service that the process is no longer interested in messages that involve the specified file.

The ToolTalk service removes this file value from any currently registered patterns and sets the default file to NULL.

The *filepath* argument is the name of the file in which the process is no longer interested.

RETURN VALUE

Upon successful completion, the *tt_file_quit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_PATH
: The specified pathname included an unsearchable directory.

SEE ALSO

<Tt/tt_c.h>, *tt_default_file*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_free — free storage from the ToolTalk API allocation stack

SYNOPSIS

```
#include <Tt/tt_c.h>

void tt_free(caddr_t p);
```

DESCRIPTION

The *tt_free*( ) function frees storage from the ToolTalk API allocation stack.

The *p* argument is the address of the storage in the ToolTalk API allocation stack to be freed.

RETURN VALUE

The *tt_free*( ) function returns no value.

APPLICATION USAGE

The application should use the *tt_free*( ) function instead of *tt_mark*( ) and *tt_release*( ) if, for example, the process is in a loop (that is, it obtains strings from the ToolTalk service and processes each in turn).

SEE ALSO

<Tt/tt_c.h>, *tt_malloc*( ), *tt_mark*( ), *tt_release*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_host_file_netfile — map between local and canonical pathnames on a remote host

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_host_file_netfile(const char *host,
                           const char *filename);
```

DESCRIPTION

The *tt_host_file_netfile*( ) function performs a conversion equivalent to that of the *tt_file_netfile*( ) function, but performs it on a remote host.

The *filename* argument is a pathname (absolute or relative) that is valid on the remote host. Every component of *filename* must exist, except for the last component. The *host* argument is a name of a remote host.

RETURN VALUE

Upon successful completion, the *tt_host_file_netfile*( ) function returns a freshly allocated null-terminated string of unspecified format, which can be passed to *tt_netfile_file*( ) or *tt_host_netfile_file*( ); otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_ERR_PATH
: The *filename* argument is a path that is not valid on the remote host.

TT_ERR_DBAVAIL
: The ToolTalk database server could not be reached on *host*, perhaps because the host is unavailable or cannot be reached through the network.

TT_ERR_DBEXIST
: The ToolTalk database server is not properly installed on *host*.

TT_ERR_UNIMP
: The ToolTalk database server contacted is of a version that does not support *tt_host_file_netfile*( ).

APPLICATION USAGE

The *tt_file_netfile*( ), *tt_netfile_file*( ), *tt_host_file_netfile*( ) and *tt_host_netfile_file*( ) functions allow an application to determine a path valid on remote hosts, perhaps for purposes of constructing a command string valid for remote execution on that host. By composing the two calls, paths for files not accessible from the current host can be constructed. For example, if path **/sample/file** is valid on host A, a program running on host B can use

```
tt_host_netfile_file("C", tt_host_file_netfile("A", "/sample/file"))
```

to determine a path to the same file valid on host C, if such a path is possible.

Allocated strings should be freed using either *tt_free*( ) or *tt_release*( ).

The *tt_open*( ) function need not be called before *tt_host_file_netfile*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_file_netfile*( ), *tt_netfile_file*( ), *tt_host_netfile_file*( ), *tt_open*( ), *tt_free*( ), *tt_release*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_host_netfile_file — map between canonical and local pathnames on a remote host

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_host_netfile_file(const char *host,
                           const char *netfilename);
```

DESCRIPTION

The *tt_host_netfile_file*( ) function performs a conversion equivalent to that of the *tt_netfile_file*( ) function, but performs it on a remote host.

The *host* argument is the host on which the file resides. The *netfilename* argument is a copy of a null-terminated string returned by *tt_netfile_file*( ) or *tt_host_netfile_file*( ).

RETURN VALUE

Upon successful completion, the *tt_host_netfile_file*( ) function returns a freshly allocated null-terminated string of unspecified format, which can be passed to *tt_host_netfile_file*( ); otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_ERR_DBAVAIL
: The ToolTalk database server could not be reached on *host*, perhaps because the host is unavailable or cannot be reached through the network.

TT_ERR_DBEXIST
: The ToolTalk database server is not properly installed on *host*.

TT_ERR_NETFILE
: The *netfilename* is not a valid netfilename.

TT_ERR_UNIMP
: The ToolTalk database server contacted is of a version that does not support *tt_host_netfile_file*( ).

APPLICATION USAGE

The *tt_file_netfile*( ), *tt_netfile_file*( ), *tt_host_file_netfile*( ) and *tt_host_netfile_file*( ) functions allow an application to determine a path valid on remote hosts, perhaps for purposes of constructing a command string valid for remote execution on that host. By composing the two calls, paths for files not accessible from the current host can be constructed. For example, if path **/sample/file** is valid on host A, a program running on host B can use

```
tt_host_netfile_file("C", tt_host_file_netfile("A", "/sample/file"))
```

to determine a path to the same file valid on host C, if such a path is possible.

Allocated strings should be freed using either *tt_free*( ) or *tt_release*( ).

The *tt_open*( ) function need not be called before *tt_host_netfile_file*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_file_netfile*( ), *tt_netfile_file*( ), *tt_host_file_netfile*( ), *tt_open*( ), *tt_free*( ), *tt_release*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_icontext_join — add an integer value to the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_icontext_join(const char *slotname, int value);
```

DESCRIPTION

The *tt_icontext_join*( ) function adds the given integer value to the list of values for the contexts of all patterns.

The context is compared to currently registered patterns for the procid. If a pattern has a slot with the specified name, the given integer value is added to the list of values for that slot.

The *slotname* argument is the name of the context. The *value* argument is the value to be added.

RETURN VALUE

Upon successful completion, the *tt_icontext_join*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_icontext_quit — remove an integer value from the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_icontext_quit(const char *slotname, int value);
```

DESCRIPTION

The *tt_icontext_quit*( ) function removes the given integer value from the list of values for the contexts of all patterns.

The context is compared to currently registered patterns for the procid. If a pattern has a slot with the specified name, the given integer value is removed from the list of values for that slot.

If there are duplicate values, only one value is removed.

The *slotname* argument is the name of the context. The *value* argument is the value to be added.

RETURN VALUE

Upon successful completion, the *tt_icontext_quit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_initial_session — return the initial session identifier

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_initial_session(void);
```

DESCRIPTION

The *tt_initial_session*( ) function returns the initial session identifier of the *ttsession* with which the current process identifier is associated.

The current process identifier is obtained by calling *tt_open*( ).

RETURN VALUE

Upon successful completion, the *tt_initial_session*( ) function returns the identifier for the current ToolTalk session. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_open*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_int_error — return the status of an error object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_int_error(int return_val);
```

DESCRIPTION

The *tt_int_error*( ) function returns the status of an error object.

The *return_val* argument is the integer returned by a ToolTalk function.

RETURN VALUE

Upon successful completion, the *tt_int_error*( ) function returns either TT_OK, if the integer is not an error object, or the encoded **Tt_status** value if the integer is an error object.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_is_err — check status value

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_is_err(Tt_status s);
```

DESCRIPTION

The *tt_is_err*( ) function checks whether a status value is a warning or an error.

The *s* argument is the **Tt_status** code to check.

RETURN VALUE

Upon successful completion, the *tt_is_err*( ) function returns one of the following integers:

0 The **Tt_status** is either a warning or TT_OK.

1 The **Tt_status** is an error.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_malloc — allocate storage on the ToolTalk API allocation stack

SYNOPSIS

```
#include <Tt/tt_c.h>

caddr_t tt_malloc(size_t s);
```

DESCRIPTION

The *tt_malloc*( ) function allocates storage on the ToolTalk API allocation stack.

The *s* argument is the amount of storage to be allocated in bytes.

RETURN VALUE

Upon successful completion, the *tt_malloc*( ) function returns the address of the storage in the ToolTalk API allocation stack that is to be allocated. If NULL is returned, no storage is available.

APPLICATION USAGE

This function allows the application-provided callback routines to take advantage of the allocation stack; for example, a query filter function can allocate storage to accumulate a result.

SEE ALSO

<Tt/tt_c.h>, *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_mark — mark a storage position in the ToolTalk API allocation stack

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_mark(void);
```

DESCRIPTION

The *tt_mark*( ) function marks a storage position in the ToolTalk API allocation stack.

RETURN VALUE

Upon successful completion, the *tt_mark*( ) function returns an integer that marks the storage position in the ToolTalk API allocation stack.

SEE ALSO

<Tt/tt_c.h>, *tt_release*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_accept — declare that the process has been initialised and can accept messages

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_accept(Tt_message m);
```

DESCRIPTION

The *tt_message_accept*( ) function declares that the process has been initialised and can accept messages.

The ToolTalk service invokes this function for start messages.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_accept*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_UNIMP
: The ToolTalk function called is not implemented.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_address — retrieve the address attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_address tt_message_address(Tt_message m);
```

DESCRIPTION

The *tt_message_address*( ) function retrieves the address attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_address*( ) function returns a value that specifies which message attributes form the address of this message. The *tt_message_address*( ) function returns one of the following **Tt_address** values:

TT_HANDLER
: The message is addressed to a specific handler that can perform this operation with these arguments.

TT_OBJECT
: The message is addressed to a specific object that can perform this operation with these arguments.

TT_OTYPE
: The message is addressed to the type of object that can perform this operation with these arguments.

TT_PROCEDURE
: The message is addressed to any process that can perform this operation with these arguments.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_address_set — set the address attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_address_set(Tt_message m, Tt_address a);
```

DESCRIPTION

The *tt_message_address_set*( ) function sets the address attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *a* argument specifies which message attributes form the address to which the message will be delivered. The following values are defined:

TT_HANDLER
: The message is addressed to a specific handler that can perform this operation with these arguments.

TT_OBJECT
: The message is addressed to a specific object that can perform this operation with these arguments.

TT_OTYPE
: The message is addressed to the type of object that can perform this operation with these arguments.

TT_PROCEDURE
: The message is addressed to any process that can perform this operation with these arguments.

RETURN VALUE

Upon successful completion, the *tt_message_address_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_add — add a new argument to a message object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_arg_add(Tt_message m,
                             Tt_mode n,
                             const char *vtype,
                             const char *value);
```

DESCRIPTION

The *tt_message_arg_add*( ) function adds a new argument to a message object.

The application must add all arguments before the message is sent. To change existing argument values, the application must use only modes TT_OUT or TT_INOUT.

Adding arguments when replying to a message produces undefined results.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument specifies who (sender, handler, observers) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The *vtype* argument describes the type of argument data being added. The *value* argument is the contents for the message argument attribute. The application can use NULL either for values of mode TT_OUT, or if the value is to be filled in later with one of the following:

tt_message_arg_val_set ( )
tt_message_barg_val_set ( )
tt_message_iarg_val_set ( )

RETURN VALUE

Upon successful completion, the *tt_message_arg_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_MODE
: The specified **Tt_mode** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_message_arg_val_set*( ), *tt_message_barg_add*( ), *tt_message_iarg_add*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_bval — retrieve the byte-array value of a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_arg_bval(Tt_message m,
                              int n,
                              unsigned char **value,
                              int *len);
```

DESCRIPTION

The *tt_message_arg_bval*( ) function retrieves the byte-array value of the *n*th message argument.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be retrieved. The first argument is numbered zero. The *value* argument is the address of a character pointer to which the ToolTalk service is to point a string that contains the contents of the argument. The *len* argument is the address of an integer to which the ToolTalk service is to set the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_message_arg_bval*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_bval_set — set the byte-array value and type of a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_arg_bval_set(Tt_message m,
                                  int n,
                                  const unsigned char *value,
                                  int len);
```

DESCRIPTION

The *tt_message_arg_bval_set*( ) function sets the byte-array value and the type of the *n*th message argument.

This function also changes the value of an existing *n*th message argument to a byte string.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to set. The first argument is numbered zero. The *value* argument is the byte string with the contents for the message argument. The *len* argument is the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_message_arg_bval_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The sending process can use *tt_message_arg_bval_set*( ) to fill in opaque data.

SEE ALSO

<Tt/tt_c.h>, *tt_message_barg_add*( ), *tt_message_arg_val_set*( ), *tt_message_arg_ival_set*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_ival — retrieve the integer value of a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_arg_ival(Tt_message m,
                              int n,
                              int *value);
```

DESCRIPTION

The *tt_message_arg_ival*( ) function retrieves the integer value of the *n*th message argument.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be retrieved. The first argument is numbered zero. The *value* argument is a pointer to an integer where the ToolTalk service is to store the contents of the argument.

RETURN VALUE

Upon successful completion, the *tt_message_arg_ival*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_ival_set — add an integer value in a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_arg_ival_set(Tt_message m,
                                  int n,
                                  int value);
```

DESCRIPTION

The *tt_message_arg_ival_set*( ) function adds an integer value in the *n*th message argument.

This function also changes the value of an existing *n*th message argument to an integer.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be set. The first argument is numbered zero. The *value* argument is the contents for the message argument.

RETURN VALUE

Upon successful completion, the *tt_message_arg_ival_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_message_iarg_add*( ), *tt_message_arg_val_set*( ), *tt_message_arg_bval_set*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_mode — return the mode of a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_mode tt_message_arg_mode(Tt_message m,
                            int n);
```

DESCRIPTION

The *tt_message_arg_mode*( ) function returns the mode of the *n*th message argument.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be returned. The first argument is numbered zero.

RETURN VALUE

Upon successful completion, the *tt_message_arg_mode*( ) function returns a value that specifies who (sender, handler, observers) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the **Tt_mode** integer return value:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_type — retrieve the type of a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_arg_type(Tt_message m,
                          int n);
```

DESCRIPTION

The *tt_message_arg_type*( ) function retrieves the type of the *n*th message argument.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be retrieved. The first argument is numbered zero.

RETURN VALUE

Upon successful completion, the *tt_message_arg_type*( ) function returns the type of the *n*th message argument. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application can use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_val — return a pointer to the value of a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_arg_val(Tt_message m,
                         int n);
```

DESCRIPTION

The *tt_message_arg_val*( ) function returns a pointer to the value of the *n*th message argument.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be returned. The first argument is numbered zero.

RETURN VALUE

Upon successful completion, the *tt_message_arg_val*( ) function returns the contents for the message argument. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application can use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_val_set — change the value of a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_arg_val_set(Tt_message m,
                                 int n,
                                 const char *value);
```

DESCRIPTION

The *tt_message_arg_val_set*( ) function changes the value of the *n*th message argument.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be changed. The first argument is numbered zero. The *value* argument is the contents for the message argument.

RETURN VALUE

Upon successful completion, the *tt_message_arg_val_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_xval — retrieve and deserialise the data from a message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_arg_xval(Tt_message m,
                              int n,
                              xdrproc_t xdr_proc,
                              void **value);
```

DESCRIPTION

The *tt_message_arg_xval*( ) function retrieves and deserialises the data from a message argument. This function uses an XDR routine that is supplied by the client.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be returned. The first argument is numbered zero. The *xdr_proc* argument points to the XDR procedure to be used to deserialise the data in the *n*th argument into newly allocated storage, the address of which will be stored in the pointer whose address is *value*.

The *value* argument is the data to be deserialised.

RETURN VALUE

Upon successful completion, the *tt_message_arg_xval*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_MODE
: The specified **Tt_mode** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length structure.

APPLICATION USAGE

The allocation calls are made by the XDR procedure; therefore, any storage allocated is not allocated from the ToolTalk allocation stack. The application should use the *xdr_free*( ) call to free this storage.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_arg_xval_set — serialise and set data into an existing message argument

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_arg_xval_set(Tt_message m,
                                  int n,
                                  xdrproc_t xdr_proc,
                                  void *value);
```

DESCRIPTION

The *tt_message_arg_xval_set*( ) function serialises and sets data into an existing message argument.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the argument to be changed. The first argument is numbered zero. The *xdr_proc* argument causes *tt_message_arg_xval_set*( ) to serialise the data pointed to by *value* and store it as a byte string value of the *n*th argument of the message. The *value* argument is the data to be serialised.

RETURN VALUE

Upon successful completion, the *tt_message_arg_xval_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_MODE
: The specified **Tt_mode** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length expression.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_args_count — return the number of arguments in the message

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_message_args_count(Tt_message m);
```

DESCRIPTION

The *tt_message_args_count*( ) function returns the number of arguments in the message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_args_count*( ) function returns the total number of arguments in the message. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_barg_add — add an argument to a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_barg_add(Tt_message m,
                              Tt_mode n,
                              const char *vtype,
                              const unsigned char *value,
                              int len);
```

DESCRIPTION

The *tt_message_barg_add*( ) function adds an argument to a pattern that may have a byte-array value that contains embedded nulls.

To change existing argument values, the application must use only modes TT_OUT or TT_INOUT.

Adding arguments when replying to a message produces undefined results.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument specifies who (sender, handler, observers) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The *vtype* argument describes the type of argument data being added.

The ToolTalk service treats the value as an opaque byte string. To pass structured data, the application and the receiving application must encode and decode these opaque byte strings. The most common method to do this is XDR.

The *value* argument is the value to be added. The *len* argument is the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_message_barg_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_message_barg_val_set*( ), *tt_message_arg_add*( ), *tt_message_iarg_add*( ); the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_bcontext_set — set the byte-array value of a message's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_bcontext_set(Tt_message m,
                                  const char *slotname,
                                  const unsigned char *value,
                                  int length);
```

DESCRIPTION

The *tt_message_bcontext_set*( ) function sets the byte-array value of a message's context.

This function overwrites any previous value associated with *slotname*.

The *m* argument is the opaque handle for the message involved in this operation. The *slotname* argument describes the slotname in this message. The *value* argument is the byte string with the contents for the message argument. The *length* argument is the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_message_bcontext_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_callback_add — register a callback function

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_callback_add(Tt_message m,
                                  Tt_message_callback f);
```

DESCRIPTION

The *tt_message_callback_add*( ) function registers a callback function to be automatically invoked by *tt_message_receive*( ) whenever a reply or other state-change to this message is returned.

The callback is defined in **<Tt/tt_c.h>**. If the callback returns TT_CALLBACK_CONTINUE, other callbacks will be run; if no callback returns TT_CALLBACK_PROCESSED, *tt_message_receive*( ) returns the message. If the callback returns TT_CALLBACK_PROCESSED, no further callbacks are invoked for this event; *tt_message_receive*( ) does not return the message.

The *m* argument is the opaque handle for the message involved in this operation. The *f* argument is the message callback to be run.

The pattern handle will be NULL if the message did not match a dynamic pattern. This is usually the case for message callbacks.

RETURN VALUE

Upon successful completion, the *tt_message_callback_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

These callbacks are invoked from *tt_message_receive*( ); the program must, therefore, call *tt_message_receive*( ) when the file descriptor returned by *tt_fd*( ) becomes active.

The application can use *tt_message_callback_add*( ) to create wrappers for ToolTalk messages. For example, a library routine can construct a request, attach a callback to the message, send the message, and process the reply in the callback. When the callback returns TT_CALLBACK_PROCESSED, the message reply is not returned to the main program; the message and reply are, therefore, completely hidden.

SEE ALSO

<Tt/tt_c.h>, *tt_message_receive*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_class — retrieve the class attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_class tt_message_class(Tt_message m);
```

DESCRIPTION

The *tt_message_class*( ) function retrieves the class attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_class*( ) function returns a value that indicates whether the sender wants an action to take place after the message is received. The *tt_message_class*( ) function returns one of the following **Tt_status** values:

TT_NOTICE
: A notice of an event. The sender does not want feedback on this message.

TT_REQUEST
: A request for some action to be taken. The sender must be notified of progress, success or failure, and must receive any return values.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the **Tt_class** integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_class_set — set the class attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_class_set(Tt_message m,
                               Tt_class c);
```

DESCRIPTION

The *tt_message_class_set*( ) function sets the class attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *c* argument indicates whether an action is to take place after the message is received. The following values are defined:

TT_NOTICE
: A notice of an event. The sender does not want feedback on this message.

TT_REQUEST
: A request for some action to be taken. The sender must be notified of progress, success or failure, and must receive any return values.

RETURN VALUE

Upon successful completion, the *tt_message_class_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_context_bval — retrieve the byte-array value and length of a message's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_context_bval(Tt_message m,
                                  const char *slotname,
                                  unsigned char **value,
                                  int *len);
```

DESCRIPTION

The *tt_message_context_bval*( ) function retrieves the byte-array value and length of a message's context.

If there is no context slot associated with *slotname*, *tt_message_context_bval*( ) returns zero in *slotname* and zero in *len*.

The *m* argument is the opaque handle for the message involved in this operation. The *slotname* argument describes the context of this message. The *value* argument points to the location to return the value. The *len* argument is the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_message_context_bval*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_context_ival — retrieve the integer value of a message's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_context_ival(Tt_message m,
                                  const char *slotname,
                                  int *value);
```

DESCRIPTION

The *tt_message_context_ival*( ) function retrieves the integer value of a message's context.

The *m* argument is the opaque handle for the message involved in this operation. The *slotname* argument describes the context of this message. The *value* argument points to the location to return the value.

If there is no context slot associated with *slotname*, *tt_message_context_ival*( ) returns a NULL pointer in **value*.

RETURN VALUE

Upon successful completion, the *tt_message_context_ival*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

TT_WRN_NOTFOUND
: The named context does not exist on the specified message.

APPLICATION USAGE

The application can use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_context_set — set the character string value of a message's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_context_set(Tt_message m,
                                 const char *slotname,
                                 const char *value);
```

DESCRIPTION

The *tt_message_context_set*( ) function sets the character string value of a message's context.

The *m* argument is the opaque handle for the message involved in this operation. The *slotname* argument describes the context of this message. This function overwrites any previous value associated with *slotname*. The *value* argument is the character string to be set.

RETURN VALUE

Upon successful completion, the *tt_message_context_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_context_slotname — return the name of a message's nth context

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_context_slotname(Tt_message m,
                                  int n);
```

DESCRIPTION

The *tt_message_context_slotname*( ) function returns the name of a message's *n*th context.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument is the number of the context to be retrieved. The first context is numbered zero.

RETURN VALUE

Upon successful completion, the *tt_message_context_slotname*( ) function returns the contents for the message argument. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application can use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_context_val — retrieve the character string of a message's context

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_context_val(Tt_message m,
                             const char *slotname);
```

DESCRIPTION

The *tt_message_context_val*( ) function retrieves the character string of a message's context.

The *m* argument is the opaque handle for the message involved in this operation. The *slotname* argument describes the context of this message.

If there is no context slot associated with *slotname*, *tt_message_context_val*( ) returns a NULL pointer.

RETURN VALUE

Upon successful completion, the *tt_message_context_val*( ) function returns the contents for the message argument. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_context_xval — retrieve and deserialise the data from a message's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_context_xval(Tt_message m,
                                  const char *slotname,
                                  xdrproc_t xdr_proc,
                                  void **value);
```

DESCRIPTION

The *tt_message_context_xval*( ) function retrieves and deserialises the data from a message's context.

The *m* argument is the opaque handle for the message involved in this operation. The *slotname* argument describes the context of this message. The *xdr_proc* argument points to the XDR procedure to be used to deserialise the data in the *n*th argument into newly allocated storage, the address of which will be stored in the pointer whose address is *value*.

The *value* argument is the data to be deserialised.

RETURN VALUE

Upon successful completion, the *tt_message_context_xval*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_MODE
: The specified **Tt_mode** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length expression.

APPLICATION USAGE

The allocation calls are made by the XDR procedure; therefore, any storage allocated is not allocated from the ToolTalk allocation stack. The application should use the *xdr_free*( ) call to free this storage.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_contexts_count — return the number of contexts in a message

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_message_contexts_count(Tt_message m);
```

DESCRIPTION

The *tt_message_contexts_count*( ) function returns the number of contexts in a message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_contexts_count*( ) function returns the total number of contexts in the message. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_create — create a new message object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_message tt_message_create(void);
```

DESCRIPTION

The *tt_message_create*( ) function creates a new message object.

The ToolTalk service returns a message handle that is an opaque pointer to a ToolTalk structure.

RETURN VALUE

Upon successful completion, the *tt_message_create*( ) function returns the unique opaque handle that identifies the message object. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The process identification is not valid.

APPLICATION USAGE

A return value of TT_ERR_PROCID implies that *tt_open*( ) was not issued before *tt_message_create*( ).

If the ToolTalk service is unable to create a message when requested, *tt_message_create*( ) returns an invalid handle. When the application attempts to use this handle with another ToolTalk function, the ToolTalk service will return TT_ERR_POINTER.

SEE ALSO

<Tt/tt_c.h>, *tt_open*( ), *tt_message_send*( ), *tt_message_destroy*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_create_super — create and re-address a copy of a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_message tt_message_create_super(Tt_message m);
```

DESCRIPTION

The *tt_message_create_super*( ) function creates a copy of the specified message and re-addresses the copy of the message to the parent of the otype contained within the message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_create_super*( ) function returns the opaque unique handle for the re-addressed message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_ADDRESS
: The specified **Tt_address** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The objid passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The otype of the message *m* can be determined using the *tt_message_otype*( ) function.

SEE ALSO

<Tt/tt_c.h>, *tt_message_otype*( ), *tt_message_send*( ), *tt_message_destroy*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_destroy — destroy a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_destroy(Tt_message m);
```

DESCRIPTION

The *tt_message_destroy*( ) function destroys the message.

Destroying a message has no effect on the delivery of a message already sent.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_destroy*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

If the application sent a request and is expecting a reply with return values, the application should destroy the message after it have received the reply. If the application sends a notice, the application can destroy the message immediately after it sends the notice.

SEE ALSO

<Tt/tt_c.h>, *tt_message_create*( ), *tt_message_create_super*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_disposition — retrieve the disposition attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_disposition tt_message_disposition(Tt_message m);
```

DESCRIPTION

The *tt_message_disposition*( ) function retrieves the disposition attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_disposition*( ) function returns a value that indicates whether an instance of the receiving process should be started to receive the message immediately, or whether the message is to be queued until the receiving process is started at a later time. The *tt_message_disposition*( ) function returns one of the following **Tt_disposition** values:

TT_DISCARD
: There is no receiver for this message. The message will be returned to the sender with the **Tt_status** field containing TT_FAILED.

TT_QUEUE
: Queue the message until a process of the proper ptype receives the message.

TT_START
: Attempt to start a process of the proper ptype if none is running.

TT_QUEUE+TT_START
: Queue the message and attempt to start a process of the proper ptype if none is running.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the **Tt_disposition** integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_disposition_set — set the disposition attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_disposition_set(Tt_message m,
                                     Tt_disposition r);
```

DESCRIPTION

The *tt_message_disposition_set*( ) function sets the disposition attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *r* argument indicates whether an instance of the receiving process is to be started to receive the message immediately, or whether the message is to be queued until the receiving process is started at a later time. The following values are defined:

TT_DISCARD
: There is no receiver for this message. The message will be returned to the sender with the **Tt_status** field containing TT_FAILED.

TT_QUEUE
: Queue the message until a process of the proper ptype receives the message.

TT_START
: Attempt to start a process of the proper ptype if none is running.

TT_QUEUE+TT_START
: Queue the message and attempt to start a process of the proper ptype if none is running.

RETURN VALUE

Upon successful completion, the *tt_message_disposition_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_fail — indicate a message cannot be handled

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_fail(Tt_message m);
```

DESCRIPTION

The *tt_message_fail*( ) function informs the ToolTalk service that the process cannot handle the request just received.

This function also informs the ToolTalk service that the message is not be offered to other processes of the same ptype. The ToolTalk service will send the message back to the sender with state TT_FAILED.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_fail*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NOTHANDLER
: This application is not the handler for this message.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

The status value must be greater than TT_ERR_LAST to avoid confusion with the ToolTalk service status values.

APPLICATION USAGE

To distinguish this case from the case where a message failed because no matching handler could be found, the application should place an explanatory message code in the status attribute of the message with *tt_message_status_set*( ) and *tt_message_status_string_set*( ) before calling *tt_message_fail*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_message_status_set*( ), *tt_message_status_string_set*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_file — retrieves the file attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_file(Tt_message m);
```

DESCRIPTION

The *tt_message_file*( ) function retrieves the file attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_file*( ) function returns a string containing the file attribute of the specified message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_file_set — set the file attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_file_set(Tt_message m,
                              const char *file);
```

DESCRIPTION

The *tt_message_file_set*( ) function sets the file attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *file* argument is the name of the file involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_file_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_FILE
: The specified file does not exist or it is inaccessible.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_gid — retrieve the group identifier attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

gid_t tt_message_gid(Tt_message m);
```

DESCRIPTION

The *tt_message_gid*( ) function retrieves the group identifier attribute from the specified message.

The ToolTalk service automatically sets the group identifier of a message with the group identifier of the process that created the message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_gid*( ) function returns the group identifier of the message. If the group **nobody** is returned, the message handle is not valid.

SEE ALSO

<Tt/tt_c.h>, *tt_message_uid*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_handler — retrieve the handler attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_handler(Tt_message m);
```

DESCRIPTION

The *tt_message_handler*( ) function retrieves the handler attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_handler*( ) function returns the character value that uniquely identifies the process that is to handle the message (**Tt_state** = TT_CREATED or TT_SENT) or the process that did handle the message (**Tt_state** = TT_SENT or TT_HANDLED). The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_handler_ptype — retrieve the handler ptype attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_handler_ptype(Tt_message m);
```

DESCRIPTION

The *tt_message_handler_ptype*( ) function retrieves the handler ptype attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_handler_ptype*( ) function returns the type of process that should handle this message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_handler_ptype_set — set the handler ptype attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_handler_ptype_set(Tt_message m,
                                       const char *ptid);
```

DESCRIPTION

The *tt_message_handler_ptype_set*( ) function sets the handler ptype attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *ptid* argument is the type of process that is to handle this message.

RETURN VALUE

Upon successful completion, the *tt_message_handler_ptype_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_handler_set — set the handler attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_handler_set(Tt_message m,
                                 const char *procid);
```

DESCRIPTION

The *tt_message_handler_set*( ) function sets the handler attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *procid* argument is the character value that uniquely identifies the process that is to handle the message.

RETURN VALUE

Upon successful completion, the *tt_message_handler_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_iarg_add — add a new argument to a message object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_iarg_add(Tt_message m,
                              Tt_mode n,
                              const char *vtype,
                              int value);
```

DESCRIPTION

The *tt_message_iarg_add*( ) function adds a new argument to a message object and sets the value to a given integer.

Add all arguments before the message is sent. To change existing argument values, the application must use only modes TT_OUT or TT_INOUT.

Adding arguments when replying to a message produces undefined results.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument specifies who (sender, handler, observers) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The *vtype* argument describes the type of argument data being added. The *value* argument is the value to be added.

RETURN VALUE

Upon successful completion, the *tt_message_iarg_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_MODE
: The specified **Tt_mode** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_message_arg_ival_set*( ), *tt_message_arg_add*( ), *tt_message_barg_add*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_icontext_set — set the integer value of a message's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_icontext_set(Tt_message m,
                                  const char *slotname,
                                  int value);
```

DESCRIPTION

The *tt_message_icontext_set*( ) function sets the integer value of a message's context.

This function overwrites any previous value associated with *slotname*.

The *m* argument is the opaque handle for the message involved in this operation. The *slotname* argument describes the context of this message. The *value* argument is the integer value to be set.

RETURN VALUE

Upon successful completion, the *tt_message_icontext_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_id — retrieve the identifier of a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_id(Tt_message m);
```

DESCRIPTION

The *tt_message_id*( ) function retrieves the identifier of the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_id*( ) function returns the character string value that uniquely identifies the message across all running ToolTalk sessions. The identifier of the message is set at its creation and never changes. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_object — retrieve the object attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_object(Tt_message m);
```

DESCRIPTION

The *tt_message_object*( ) function retrieves the object attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_object*( ) function returns the object involved in this message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The objid passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_object_set — set the object attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_object_set(Tt_message m,
                                const char *objid);
```

DESCRIPTION

The *tt_message_object_set*( ) function sets the object attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *objid* argument is the identifier of the specified object.

RETURN VALUE

Upon successful completion, the *tt_message_object_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_op — retrieve the operation attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_op(Tt_message m);
```

DESCRIPTION

The *tt_message_op*( ) function retrieves the operation attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_op*( ) function returns the operation which the receiving process is to perform. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_op_set — set the operation attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_op_set(Tt_message m,
                            const char *opname);
```

DESCRIPTION

The *tt_message_op_set*( ) function sets the operation attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *opname* argument is the operation that the receiving process is to perform.

RETURN VALUE

Upon successful completion, the *tt_message_op_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_opnum — retrieve the operation number attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_message_opnum(Tt_message m);
```

DESCRIPTION

The *tt_message_opnum*( ) function retrieves the operation number attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_opnum*( ) function returns the number of the operation involved in this message. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_otype — retrieve the object type attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_otype(Tt_message m);
```

DESCRIPTION

The *tt_message_otype*( ) function retrieves the object type attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_otype*( ) function returns the type of the object involved in this message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_otype_set — set the otype attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_otype_set(Tt_message m,
                               const char *otype);
```

DESCRIPTION

The *tt_message_otype_set*( ) function sets the object type (*otype*) attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *otype* argument is the type of the object involved in this message.

RETURN VALUE

Upon successful completion, the *tt_message_otype_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_pattern — return the pattern matched by a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_pattern tt_message_pattern(Tt_message m);
```

DESCRIPTION

The *tt_message_pattern*( ) function returns the pattern that the specified message matched.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_pattern*( ) function returns the opaque handle for a message pattern. The application can use *tt_ptr_error*( ) to determine if the handle is valid. The *tt_message_pattern*( ) function returns one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_print — format a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_print(Tt_message *m);
```

DESCRIPTION

The *tt_message_print*( ) function formats a message in the same way a message is formatted for the *ttsession* trace and returns a string containing it.

The *m* argument is the message to be formatted.

RETURN VALUE

Upon successful completion, the *tt_message_print*( ) function returns the formatted string. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The *tt_message_print*( ) function allows an application to dump out messages that are received but not understood.

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_free*( ), *tt_ptr_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_receive — receive a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_message tt_message_receive(void);
```

DESCRIPTION

The *tt_message_receive*( ) function returns a handle for the next message queued to be delivered to the process and also runs any message or pattern callbacks applicable to the queued message.

If the return value of *tt_message_status*( ) for this message is TT_WRN_START_MESSAGE, the ToolTalk service started the process to deliver the queued message; the process must reply to this message. If the return value of *tt_message_receive*( ) is zero, no message is available.

RETURN VALUE

Upon successful completion, the *tt_message_receive*( ) function returns the handle for the message object. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

APPLICATION USAGE

A zero value can occur if a message or pattern callback processes the message. It can also occur if the interval is too long between the time the file descriptor became active and the *tt_message_receive*( ) call was made. In the latter case, the ToolTalk service will time out and offer the message to another process.

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_reject — reject a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_reject(Tt_message m);
```

DESCRIPTION

The *tt_message_reject*( ) function informs the ToolTalk service that the process cannot handle this message. The ToolTalk service will attempt to deliver the message to other handlers.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_reject*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NOTHANDLER
: This application is not the handler for this message.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_reply — reply to a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_reply(Tt_message m);
```

DESCRIPTION

The *tt_message_reply*( ) function informs the ToolTalk service that the process has handled the message and filled in all return values.

The ToolTalk service sends the message back to the sending process and fills in the state attribute with TT_HANDLED.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_reply*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NOTHANDLER
: This application is not the handler for this message.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_scope — retrieve the scope attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_scope tt_message_scope(Tt_message m);
```

DESCRIPTION

The *tt_message_scope*( ) function retrieves the scope attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_scope*( ) function returns a value that identifies the set of processes eligible to receive the message. The following values are defined:

TT_SESSION
: All processes joined to the indicated session are eligible.

TT_FILE
: All processes joined to the indicated file are eligible.

TT_BOTH
: All processes joined to either indicated file or the indicated session are eligible.

TT_FILE_IN_SESSION
: All processes joined to both the indicated file and the indicated session are eligible.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the **Tt_scope** integer return value:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_scope_set — set the scope attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_scope_set(Tt_message m,
                               Tt_scope s);
```

DESCRIPTION

The *tt_message_scope_set*( ) function sets the scope attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *s* argument identifies the set of processes eligible to receive the message. The following values are defined:

TT_SESSION
: All processes joined to the indicated session are eligible.

TT_FILE
: All processes joined to the indicated file are eligible.

TT_BOTH
: All processes joined to either indicated file or the indicated session are eligible.

TT_FILE_IN_SESSION
: All processes joined to both the indicated file and the indicated session are eligible.

RETURN VALUE

Upon successful completion, the *tt_message_scope_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_send — send a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_send(Tt_message m);
```

DESCRIPTION

The *tt_message_send*( ) function sends the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_send*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_ADDRESS
: The specified **Tt_address** is invalid.

TT_ERR_CLASS
: The specified **Tt_class** is invalid.

TT_ERR_FILE
: The specified file does not exist or it is inaccessible.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The objid passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

TT_WRN_STALE_OBJID
: The object attribute in the message has been replaced with a newer one.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_send_on_exit — set up a message to send upon unexpected exit

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_send_on_exit(Tt_message m);
```

DESCRIPTION

The *tt_message_send_on_exit*( ) function requests that the ToolTalk service send this message if the process exits unexpectedly. The message is sent to the ToolTalk service, which queues the message internally until either of two events occur:

1. The procid that sent the *tt_message_send_on_exit*( ) message to the ToolTalk service calls *tt_close*( ). In this case, the queued message is deleted.
2. The connection between the *ttsession* server and the process that sent the *tt_message_send_on_exit*( ) message to the ToolTalk service is broken; for example, if the application has crashed.

In this case, the ToolTalk service matches the queued message to its patterns and delivers it in the same manner as if the process had sent the message normally before exiting.

If a process sends a normal termination message but exits without calling *tt_close*( ), both the normal termination message and the on_exit message are delivered.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_send_on_exit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_ADDRESS
: The specified **Tt_address** is invalid.

TT_ERR_CLASS
: The specified **Tt_class** is invalid.

TT_ERR_FILE
: The specified file does not exist or it is inaccessible.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The objid passed to the ToolTalk service does not reference an existing object spec.

SEE ALSO

<Tt/tt_c.h>, *tt_close*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_sender — retrieve the sender attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_sender(Tt_message m);
```

DESCRIPTION

The *tt_message_sender*( ) function retrieves the sender attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_sender*( ) function returns the character value that uniquely identifies the sending process. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_sender_ptype — retrieve the sender ptype attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_sender_ptype(Tt_message m);
```

DESCRIPTION

The *tt_message_sender_ptype*( ) function retrieves the sender ptype attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_sender_ptype*( ) function returns the sending process. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_sender_ptype_set — set the sender ptype attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_sender_ptype_set(Tt_message m,
                                      const char *ptid);
```

DESCRIPTION

The *tt_message_sender_ptype_set*( ) function sets the sender ptype attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *ptid* argument is the type of process that is sending this message.

RETURN VALUE

Upon successful completion, the *tt_message_sender_ptype_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_session — retrieve the session attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_session(Tt_message m);
```

DESCRIPTION

The *tt_message_session*( ) function retrieves the session attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_session*( ) function returns the identifier of the session to which this message applies. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_session_set — set the session attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_session_set(Tt_message m,
                                 const char *sessid);
```

DESCRIPTION

The *tt_message_session_set*( ) function sets the session attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *sessid* argument is the identifier of the session in which the process is interested.

RETURN VALUE

Upon successful completion, the *tt_message_session_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_state — retrieve the state attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_state tt_message_state(Tt_message m);
```

DESCRIPTION

The *tt_message_state*( ) function retrieves the state attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_state*( ) function returns a value that indicates the current delivery state of the message. The *tt_message_state*( ) function returns one of the following **Tt_status** values:

TT_CREATED
: The message has been created, but not yet sent.

TT_SENT
: The message has been sent, but not yet handled.

TT_HANDLED
: The message has been handled; return values are valid.

TT_FAILED
: The message could not be delivered to a handler.

TT_QUEUED
: The message has been queued for delivery.

TT_STARTED
: The ToolTalk service is attempting to start a process to handle the message.

TT_REJECTED
: The message has been rejected by a possible handler.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the **Tt_state** integer return value:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_status — retrieve the status attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_message_status(Tt_message m);
```

DESCRIPTION

The *tt_message_status*( ) function retrieves the status attribute from the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_status*( ) function returns an integer that describes the status stored in the status attribute of this message. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_message_status_string*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_status_set — set the status attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_status_set(Tt_message m,
                                int status);
```

DESCRIPTION

The *tt_message_status_set*( ) function sets the status attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *status* argument is the status to be stored in this message.

RETURN VALUE

Upon successful completion, the *tt_message_status_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

The status value must be greater than TT_ERR_LAST to avoid confusion with the ToolTalk service status values.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_status_string — retrieve the character string stored with the status attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_message_status_string(Tt_message m);
```

DESCRIPTION

The *tt_message_status_string*( ) function retrieves the character string stored with the status attribute for the specified message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_status_string*( ) function returns the status string stored in this message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_message_status*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_status_string_set — set a character string with the status attribute for a message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_status_string_set(Tt_message m,
                                       const char *status_str);
```

DESCRIPTION

The *tt_message_status_string_set*( ) function sets status string of the specified message.

The *m* argument is the opaque handle for the message involved in this operation. The *status_str* argument is the status string to be stored in this message.

RETURN VALUE

Upon successful completion, the *tt_message_status_string_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The status string should be used by the application developer to amplify on, for example, why the application is failing a message.

SEE ALSO

<Tt/tt_c.h>, *tt_message_status_set*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_uid — retrieve the user identifier attribute from a message

SYNOPSIS

```
#include <Tt/tt_c.h>

uid_t tt_message_uid(Tt_message m);
```

DESCRIPTION

The *tt_message_uid*( ) function retrieves the user identifier attribute from the specified message.

The ToolTalk service automatically sets the user identifier of a message with the user identifier of the process that created the message.

The *m* argument is the opaque handle for the message involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_message_uid*( ) function returns the user identifier of the message. If the group **nobody** is returned, the message handle is not valid.

SEE ALSO

<Tt/tt_c.h>, *tt_message_gid*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_user — retrieve the user information associated with a message object

SYNOPSIS

```
#include <Tt/tt_c.h>

void *tt_message_user(Tt_message m,
                      int key);
```

DESCRIPTION

The *tt_message_user*( ) function retrieves the user information stored in data cells associated with the specified message object.

The user data is part of the message object (that is, the storage buffer in the application); it is not a part of the actual message. The application can, therefore, only retrieve user information that the application placed in the message.

The *m* argument is the opaque handle for the message involved in this operation. The *key* argument is the user data cell to be retrieved. The user data cell must be unique for this message.

RETURN VALUE

Upon successful completion, the *tt_message_user*( ) function returns the data cell, a piece of arbitrary user data that can hold a **void** *. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned data:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

The user data cell is intended to hold an address. If the address selected is equal to one of the **Tt_status** enumerated values, the result of the *tt_ptr_error*( ) function will not be reliable.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_user_set — stores user information associated with a message object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_user_set(Tt_message m,
                              int key,
                              void *v);
```

DESCRIPTION

The *tt_message_user_set*( ) function stores user information in data cells associated with the specified message object.

The user data is part of the message object (that is, the storage buffer in the application); it is not part of the actual message. Data stored by the sending process in user data cells is not seen by handlers and observers. The application can use arguments for data that needs to be seen by handlers or observers.

The *m* argument is the opaque handle for the message involved in this operation. The *key* argument is the user data cell in which user information is to be stored. The *v* argument is the data cell, a piece of arbitrary user data that can hold a **void** *.

RETURN VALUE

Upon successful completion, the *tt_message_user_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>, *tt_message_arg_add*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_xarg_add — add an argument with an XDR-interpreted value to a message object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_xarg_add(Tt_message m,
                              Tt_mode n,
                              const char *vtype,
                              xdrproc_t xdr_proc,
                              void *value);
```

DESCRIPTION

The *tt_message_xarg_add*( ) function adds an argument with an XDR-interpreted value to a message object.

To change existing argument values, the application must use only modes TT_OUT or TT_INOUT.

Adding arguments when replying to a message produces undefined results.

The *m* argument is the opaque handle for the message involved in this operation. The *n* argument specifies who (sender, handler, observers) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The *vtype* argument describes the type of argument data being added. The *xdr_proc* argument points to the XDR procedure to be used to serialise the data pointed to by *value*. The *value* argument is the data to be serialised.

RETURN VALUE

Upon successful completion, the *tt_message_xarg_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_MODE
: The specified **Tt_mode** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length expression.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_xcontext_join — add an XDR-interpreted byte-array to the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_xcontext_join(const char *slotname,
                                   xdrproc_t xdr_proc,
                                   void *value);
```

DESCRIPTION

The *tt_message_xcontext_join*( ) function adds the given XDR-interpreted byte-array value to the list of values for the named contexts of all patterns.

The *slotname* argument describes the slotname in this message. The *xdr_proc* argument points to the XDR procedure to be used to serialise the data pointed to by *value*. The *value* argument is the data to be serialised.

RETURN VALUE

Upon successful completion, the *tt_message_xcontext_join*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length expression.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_message_xcontext_set — set the XDR-interpreted byte-array value of a message's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_message_xcontext_set(Tt_message m,
                                  const char *slotname,
                                  xdrproc_t xdr_proc,
                                  void *value);
```

DESCRIPTION

The *tt_message_xcontext_set*( ) function sets the XDR-interpreted byte-array value of a message's context.

The *m* argument is the opaque handle for the message involved in this operation. The *slotname* argument describes the slotname in this message. The *value* argument is the byte string with the contents for the message argument. The *xdr_proc* argument points to the XDR procedure to be used to serialise the data pointed to by *value*. The *value* argument is the data to be serialised.

RETURN VALUE

Upon successful completion, the *tt_message_xcontext_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer does not point at an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length expression.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_netfile_file — map between canonical and local pathnames on the local host

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_netfile_file(const char *netfilename);
```

DESCRIPTION

The *tt_netfile_file*( ) function converts a *netfilename* of the format returned by *tt_file_netfile*( ) to a pathname that is valid on the local host. If the file is not currently mounted on the local host, *tt_netfile_file*( ) constructs a pathname of the form:

/*mountpoint* / *host* / *filepath*

where *mountpoint* is the mount point pathname in the environment variable *DTMOUNTPOINT*, or **/net** if the variable is null or unset.

The *netfilename* argument is a copy of a null-terminated string returned by *tt_netfile_file*( ) or *tt_host_netfile_file*( ).

RETURN VALUE

Upon successful completion, the *tt_netfile_file*( ) function returns a null-terminated local filename; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_ERR_NETFILE
: The *netfilename* argument is not a valid netfilename.

APPLICATION USAGE

The *tt_file_netfile*( ), *tt_netfile_file*( ), *tt_host_file_netfile*( ) and *tt_host_netfile_file*( ) functions allow an application to determine a path valid on remote hosts, perhaps for purposes of constructing a command string valid for remote execution on that host. By composing the two calls, paths for files not accessible from the current host can be constructed. For example, if path **/sample/file** is valid on host A, a program running on host B can use

```
tt_host_netfile_file("C", tt_host_file_netfile("A", "/sample/file"))
```

to determine a path to the same file valid on host C, if such a path is possible.

The *netfilename* string input to *tt_netfile_file*( ) should be considered opaque; the content and format of the strings are not a public interface. These strings can be safely copied (with *strcpy*( ) or similar methods), written to files, or transmitted to other processes, perhaps on other hosts.

The *mountpoint* value is intended to be the mount point for the automounter's host map on those systems supporting automounting services.

Allocated strings should be freed using either *tt_free*( ) or *tt_release*( ).

The *tt_open*( ) function need not be called before *tt_netfile_file*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_file_netfile*( ), *tt_host_file_netfile*( ), *tt_host_netfile_file*( ), *tt_open*( ), *tt_free*( ), *tt_release*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_objid_equal — test whether two objids are equal

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_objid_equal(const char *objid1,
                   const char *objid2);
```

DESCRIPTION

The *tt_objid_equal*( ) function tests whether two objids are equal.

The *tt_objid_equal*( ) function is recommended rather than *strcmp*( ) for this purpose because the *tt_objid_equal*( ) function returns 1 even in the case where one objid is a forwarding pointer for the other.

The *objid1* argument is the identifier of the first object involved in this operation. The *objid2* argument is the identifier of the second object involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_objid_equal*( ) function returns an integer that indicates whether the objids are equal. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

0 The *objid1* and *objid2* objects are not equal.

1 The *objid1* and *objid2* objects are equal.

The application can use *tt_int_error*( ) to determine if the integer is valid. The *tt_objid_equal*( ) function returns one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The objid passed to the ToolTalk service does not reference an existing object spec.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_objid_objkey — return the unique key of an objid

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_objid_objkey(const char *objid);
```

DESCRIPTION

The *tt_objid_objkey*( ) function returns the unique key of an objid.

The *objid* argument is the identifier of the object involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_objid_objkey*( ) function returns the unique key of the *objid*. No two objids have the same unique key. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_onotice_create — create a notice

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_message tt_onotice_create(const char *objid,
                             const char *op);
```

DESCRIPTION

The *tt_onotice_create*( ) function creates a message. The created message contains the following:

Tt_address = TT_OBJECT
Tt_class = TT_NOTICE

The application can use the returned handle to add arguments and other attributes, and to send the message.

The *objid* argument is the identifier of the specified object. The *op* argument is the operation to be performed by the receiving process.

RETURN VALUE

Upon successful completion, the *tt_onotice_create*( ) function returns the unique handle that identifies the message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_NOMP
The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
The specified process identifier is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_open — return the process identifier for the calling process

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_open(void);
```

DESCRIPTION

The *tt_open*( ) function returns the process identifier for the calling process.

RETURN VALUE

The *tt_open*( ) function also sets this identifier as the default procid for the process. The *tt_open*( ) function is typically the first ToolTalk function called by a process.

The application must call *tt_open*( ) before other *tt_* calls are made. However, there are two exceptions: *tt_default_session_set*( ) and *tt_X_session*( ) can be called before *tt_open*( ).

A process can call *tt_open*( ) more than once to obtain multiple procids. To open another session, the process must make the following calls in the order specified:

tt_default_session_set ( )
tt_open ( )

RETURN VALUE

Upon successful completion, the *tt_open*( ) function returns the character value that uniquely identifies the process. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
The *ttsession* process is not running and the ToolTalk service cannot restart it.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

Each procid has its own associated file descriptor, and can join another session. To switch to another procid, the application should call *tt_default_procid_set*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_fd*( ), *tt_default_procid*( ), *tt_default_procid_set*( ), *tt_default_session*( ), *tt_default_session_set*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_orequest_create — create a request message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_message tt_orequest_create(const char *objid,
                              const char *op);
```

DESCRIPTION

The *tt_orequest_create*( ) function creates a message. The created message contains the following:

Tt_address = TT_OBJECT
Tt_class = TT_REQUEST

The application can use the returned handle to add arguments and other attributes, and to send the message.

The *objid* argument is the identifier of the specified object. The *op* argument is the operation to be performed by the receiving process.

RETURN VALUE

Upon successful completion, the *tt_orequest_create*( ) function returns the unique handle that identifies the message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_NOMP
The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
The specified process identifier is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_base — return the base otype of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_otype_base(const char *otype);
```

DESCRIPTION

The *tt_otype_base*( ) function returns the base otype of the given otype, or NULL if the given otype is not derived.

The *otype* argument is the object type involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_otype_base*( ) function returns the name of the base otype; if the given otype is not derived, this value is NULL. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_is_derived*( ), *tt_otype_derived*( ), *tt_otype_deriveds_count*( ), *tt_spec_type*( ), *tt_message_otype*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_derived — return the ith otype derived from the given otype

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_otype_derived(const char *otype,
                       int i);
```

DESCRIPTION

The *tt_otype_derived*( ) function returns the *i*th otype derived from the given otype.

The *otype* argument is the object type involved in this operation. The *i* argument is the zero-based index into the otypes derived from the given otype.

RETURN VALUE

Upon successful completion, the *tt_otype_derived*( ) function returns the name of the *i*th otype derived from the given otype. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_is_derived*( ), *tt_otype_base*( ), *tt_otype_deriveds_count*( ), *tt_spec_type*( ), *tt_message_otype*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_deriveds_count — return the number of otypes derived from an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_otype_deriveds_count(const char *otype);
```

DESCRIPTION

The *tt_otype_deriveds_count*( ) function returns the number of otypes derived from the given otype.

The *otype* argument is the object type involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_otype_deriveds_count*( ) function returns the number of otypes derived from the given otype. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_is_derived*( ), *tt_otype_base*( ), *tt_otype_derived*( ), *tt_spec_type*( ), *tt_message_otype*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_hsig_arg_mode — return the mode of an argument of a request signature of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_mode tt_otype_hsig_arg_mode(const char *otype,
                               int sig,
                               int arg);
```

DESCRIPTION

The *tt_otype_hsig_arg_mode*( ) function returns the mode of the *arg*th argument of the *sig*th request signature of the given otype.

The *otype* argument is the object type involved in this operation. The *sig* argument is the zero-based index into the request signatures of the specified otype. The *arg* argument is the zero-based index into the arguments of the specified signature.

RETURN VALUE

Upon successful completion, the *tt_otype_hsig_arg_mode*( ) function returns a value that determines who (sender or handler) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the **Tt_mode** integer return value:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_hsig_arg_type*( ), *tt_otype_hsig_count*( ), *tt_otype_hsig_args_count*( ), *tt_otype_hsig_op*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_hsig_arg_type — return the data type of an argument of a request signature of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_otype_hsig_arg_type(const char *otype,
                             int sig,
                             int arg);
```

DESCRIPTION

The *tt_otype_hsig_arg_type*( ) function returns the data type of the *arg*th argument of the *sig*th request signature of the given otype.

The *otype* argument is the object type involved in this operation. The *sig* argument is the zero-based index into the request signatures of the specified otype. The *arg* argument is the zero-based index into the arguments of the specified signature.

RETURN VALUE

Upon successful completion, the *tt_otype_hsig_arg_type*( ) function returns the data type of the specified argument. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_hsig_arg_mode*( ), *tt_otype_hsig_count*( ), *tt_otype_hsig_args_count*( ), *tt_otype_hsig_op*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_hsig_args_count — return the number of arguments of a request signature of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_otype_hsig_args_count(const char *otype,
                             int sig);
```

DESCRIPTION

The *tt_otype_hsig_args_count*( ) function returns the number of arguments of the *sig*th request signature of the given otype.

The *otype* argument is the object type involved in this operation. The *sig* argument is the zero-based index into the request signatures of the specified otype.

RETURN VALUE

Upon successful completion, the *tt_otype_hsig_args_count*( ) function returns the number of arguments of the *sig*th request signature of the given otype. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_hsig_arg_type*( ), *tt_otype_hsig_arg_mode*( ), *tt_otype_hsig_count*( ), *tt_otype_hsig_op*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_hsig_count — return the number of request signatures for an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_otype_hsig_count(const char *otype);
```

DESCRIPTION

The *tt_otype_hsig_count*( ) function returns the number of request signatures for the given otype.

The *otype* argument is the object type involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_otype_hsig_count*( ) function returns the number of request signatures for the given otype. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_hsig_arg_type*( ), *tt_otype_hsig_arg_mode*( ), *tt_otype_hsig_args_count*( ), *tt_otype_hsig_op*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_hsig_op — return the operation name of a request signature of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_otype_hsig_op(const char *otype,
                       int sig);
```

DESCRIPTION

The *tt_otype_hsig_op*( ) function returns the operation name of the *sig*th request signature of the given otype.

The *otype* argument is the object type involved in this operation. The *sig* argument is the zero-based index into the request signatures of the given otype.

RETURN VALUE

Upon successful completion, the *tt_otype_hsig_op*( ) function returns the operation attribute of the specified request signature. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_hsig_arg_type*( ), *tt_otype_hsig_arg_mode*( ), *tt_otype_hsig_args_count*( ), *tt_otype_hsig_count*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_is_derived — indicate the otype derivations

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_otype_is_derived(const char *derivedotype,
                        const char *baseotype);
```

DESCRIPTION

The *tt_otype_is_derived*( ) function specifies whether the derived otype is derived directly or indirectly from the base otype.

The *derivedotype* argument is the specified derived otype. The *baseotype* argument is the specified base otype.

RETURN VALUE

Upon successful completion, the *tt_otype_is_derived*( ) function returns 1 if the *derivedotype* is derived directly or indirectly from *baseotype*; otherwise, it returns zero.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_deriveds_count*( ), *tt_otype_base*( ), *tt_otype_derived*( ), *tt_spec_type*( ), *tt_message_otype*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_opnum_callback_add — return a callback if two opnums are equal

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_otype_opnum_callback_add(const char *otid,
                                      int opnum,
                                      Tt_message_callback f);
```

DESCRIPTION

The *tt_otype_opnum_callback_add* ( ) function adds a callback that is automatically invoked when a message is delivered because it matched a pattern derived from a signature in the named otype with an opnum equal to the specified one. The callback is defined in **<Tt/tt_c.h>**.

The *otid* argument is the identifier of the object type involved in this operation. The *opnum* argument is the opnum of the specified otype. The *f* argument is the message callback to be run.

RETURN VALUE

Upon successful completion, the *tt_otype_opnum_callback_add* ( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

APPLICATION USAGE

The *tt_otype_opnum_callback_add* ( ) function will only be called for messages delivered by virtue of matching handler signatures. The callback cannot be called for observer signatures because the observer ptype is not recorded in the incoming message.

SEE ALSO

<Tt/tt_c.h>, *tt_message_callback_add* ( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_osig_arg_mode — return the mode of an argument of a notice signature of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_mode tt_otype_osig_arg_mode(const char *otype,
                               int sig,
                               int arg);
```

DESCRIPTION

The *tt_otype_osig_arg_mode*( ) function returns the mode of the *arg*th argument of the *sig*th notice signature of the given otype.

The *otype* argument is the object type involved in this operation. The *sig* argument is the zero-based index into the notice signatures of the specified otype. The *arg* argument is the zero-based index into the arguments of the specified signature.

RETURN VALUE

Upon successful completion, the *tt_otype_osig_arg_mode*( ) function returns a value that determines who (sender or handler) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the **Tt_mode** integer return value:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_osig_arg_type*( ), *tt_otype_osig_count*( ), *tt_otype_osig_args_count*( ), *tt_otype_osig_op*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_osig_arg_type — return the data type of an argument of a notice signature of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_otype_osig_arg_type(const char *otype,
                             int sig,
                             int arg);
```

DESCRIPTION

The *tt_otype_osig_arg_type*( ) function returns the data type of the *arg*th argument of the *sig*th notice signature of the given otype.

The *otype* argument is the object type involved in this operation. The *sig* argument is the zero-based index into the notice signatures of the specified otype. The *arg* argument is the zero-based index into the arguments of the specified signature.

RETURN VALUE

Upon successful completion, the *tt_otype_osig_arg_type*( ) function returns the data type of the specified argument. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_osig_arg_mode*( ), *tt_otype_osig_count*( ), *tt_otype_osig_args_count*( ), *tt_otype_osig_op*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_osig_args_count — returns the number of arguments of a notice signature of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_otype_osig_args_count(const char *otype,
                             int sig);
```

DESCRIPTION

The *tt_otype_osig_args_count*( ) function returns the number of arguments of the *sig*th notice signature of the given otype.

The *otype* argument is the object type involved in this operation. The *sig* argument is the zero-based index into the notice signatures of the specified otype.

RETURN VALUE

Upon successful completion, the *tt_otype_osig_args_count*( ) function returns the number of arguments of the *sig*th notice signature of the given otype. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_osig_arg_type*( ), *tt_otype_osig_arg_mode*( ), *tt_otype_osig_count*( ), *tt_otype_osig_op*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_osig_count — return the number of notice signatures for an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_otype_osig_count(const char*otype);
```

DESCRIPTION

The *tt_otype_osig_count*( ) function returns the number of notice signatures for the given otype.

The *otype* argument is the object type involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_otype_osig_count*( ) function returns the number of notice signatures for the given otype. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_osig_arg_type*( ), *tt_otype_osig_arg_mode*( ), *tt_otype_osig_args_count*( ), *tt_otype_osig_op*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_otype_osig_op — return the op name of a notice signature of an otype

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_otype_osig_op(const char *otype,
                       int sig);
```

DESCRIPTION

The *tt_otype_osig_op*( ) function returns the op name of the *sig*th notice signature of the given otype.

The *otype* argument is the object type involved in this operation. The *sig* argument is the zero-based index into the notice signatures of the given otype.

RETURN VALUE

Upon successful completion, the *tt_otype_osig_op*( ) function returns the operation attribute of the specified notice signature. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_otype_osig_arg_type*( ), *tt_otype_osig_arg_mode*( ), *tt_otype_osig_args_count*( ), *tt_otype_osig_count*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_address_add — add a value to the address field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_address_add(Tt_pattern p,
                                 Tt_address d);
```

DESCRIPTION

The *tt_pattern_address_add*( ) function adds a value to the address field for the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after a *tt_pattern_create*( ) call has been made.

The *d* argument specifies which pattern attributes form the address that messages will be matched against. The following values are defined:

TT_HANDLER
: The message is addressed to a specific handler that can perform this operation with these arguments.

TT_OBJECT
: The message is addressed to a specific object that can perform this operation with these arguments.

TT_OTYPE
: The message is addressed to the type of object that can perform this operation with these arguments.

TT_PROCEDURE
: The message is addressed to any process that can perform this operation with these arguments.

RETURN VALUE

Upon successful completion, the *tt_pattern_address_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_arg_add — add an argument to a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_arg_add(Tt_pattern p,
                             Tt_mode n,
                             const char *vtype,
                             const char *value);
```

DESCRIPTION

The *tt_pattern_arg_add*( ) function adds an argument to a pattern. The application must add pattern arguments before it registers the pattern with the ToolTalk service.

The *p* argument is the opaque handle for the pattern involved in this operation The *n* argument specifies who (sender, handler, observers) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The *vtype* argument describes the type of argument data being added. The type ALL matches any argument value type. The *value* argument is the value to fill in. This value must be an unsigned character string. A NULL matches any value.

RETURN VALUE

Upon successful completion, the *tt_pattern_arg_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_register*( ), *tt_pattern_barg_add*( ), *tt_pattern_iarg_add*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_barg_add — add an argument with a value that contains embedded nulls to a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_barg_add(Tt_pattern m,
                              Tt_mode n,
                              const char *vtype,
                              const unsigned char *value,
                              int len);
```

DESCRIPTION

The *tt_pattern_barg_add*( ) function adds an argument with a value that contains embedded nulls to a pattern.

The *m* argument is the opaque handle for the pattern involved in this operation. The *n* argument specifies who (sender, handler, observers) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The *vtype* argument describes the type of argument data being added. Type **ALL** matches any argument value type.

The ToolTalk service treats the value as an opaque byte string. To pass structured data, the application and the receiving application must encode and decode these unique values. The most common method to use is XDR.

The *value* argument is the value to be added. NULL matches any value.

The *len* argument is the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_pattern_barg_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_register*( ), *tt_pattern_arg_add*( ), *tt_pattern_iarg_add*( ); the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_bcontext_add — add a byte-array value to the values in this pattern's named context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_bcontext_add(Tt_pattern p,
                                  const char *slotname,
                                  const unsigned char *value,
                                  int length);
```

DESCRIPTION

The *tt_pattern_bcontext_add*( ) function adds a byte-array value to the values in this pattern's named context.

The *p* argument is the opaque handle for the pattern involved in this operation. The *slotname* argument describes the context for this pattern. The *value* argument is the byte string with the contents for the message context. The *length* argument is the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_pattern_bcontext_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_UNIMP
: The ToolTalk function called is not implemented.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_callback_add — register a message-matching callback function

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_callback_add(Tt_pattern m,
                                  Tt_message_callback f);
```

DESCRIPTION

The *tt_pattern_callback_add*( ) function registers a callback function that will be automatically invoked by *tt_message_receive*( ) whenever a message matches the pattern.

The callback is defined in **<Tt/tt_c.h>**. If the callback returns TT_CALLBACK_CONTINUE, other callbacks will be run; if no callback returns TT_CALLBACK_PROCESSED, *tt_message_receive*( ) returns the message. If the callback returns TT_CALLBACK_PROCESSED, no further callbacks will be invoked for this event; *tt_message_receive*( ) does not return the message.

The *m* argument is the opaque handle for the pattern involved in this operation.

The *f* argument is the message callback to be run.

RETURN VALUE

Upon successful completion, the *tt_pattern_callback_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_register*( ), *tt_message_receive*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_category — return the category value of a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_category tt_pattern_category(Tt_pattern p);
```

DESCRIPTION

The *tt_pattern_category*( ) function returns the category value of the specified pattern.

The *p* argument is the opaque handle for a message pattern.

RETURN VALUE

Upon successful completion, the *tt_pattern_category*( ) function returns a value that indicates whether the receiving process will observe or handle messages. The *tt_pattern_category*( ) function returns one of the following **Tt_status** values:

TT_OBSERVE
: The receiving process will observe messages.

TT_HANDLE
: The receiving process will handle messages.

The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the **Tt_category** integer return value:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_category_set*( ), *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_category_set — fill in the category field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_category_set(Tt_pattern p,
                                  Tt_category c);
```

DESCRIPTION

The *tt_pattern_category_set*( ) function fills in the category field for the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called.

The *c* argument indicates whether the receiving process will observe or handle messages. The following values are defined:

TT_OBSERVE
: The receiving process will observe messages.

TT_HANDLE
: The receiving process will handle messages.

RETURN VALUE

Upon successful completion, the *tt_pattern_category_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_CATEGORY
: The pattern object has no category set.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_category*( ), *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_class_add — add a value to the class information for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_class_add(Tt_pattern p,
                               Tt_class c);
```

DESCRIPTION

The *tt_pattern_class_add*( ) function adds a value to the class information for the specified pattern.

If the class is TT_REQUEST, the sending process expects a reply to the message.

If the class is TT_NOTICE, the sending process does not expect a reply to the message.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *c* argument indicates whether the receiving process is to take action after the message is received. The following values are defined:

TT_NOTICE
: A notice of an event. The sender does not want feedback on this message.

TT_REQUEST
: A request for some action to be taken. The sender must be notified of progress, success or failure, and must receive any return values.

RETURN VALUE

Upon successful completion, the *tt_pattern_class_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_context_add — add a string value to the values of this pattern's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_context_add(Tt_pattern p,
                                 const char *slotname,
                                 const char *value);
```

DESCRIPTION

The *tt_pattern_context_add*( ) function adds a string value to the values of this pattern's context.

If the value pointer is NULL, a slot is created with the specified name but no value is added.

The *p* argument is the opaque handle for the pattern involved in this operation. The *slotname* argument describes the context of this pattern. The *value* argument is the value to be added.

RETURN VALUE

Upon successful completion, the *tt_pattern_context_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_UNIMP
: The ToolTalk function called is not implemented.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_create — request a new pattern object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_pattern tt_pattern_create(void);
```

DESCRIPTION

The *tt_pattern_create*( ) function requests a new pattern object.

After receiving the pattern object, the application fills in the message pattern fields to indicate what type of messages the process wants to receive and then registers the pattern with the ToolTalk service.

The application can supply multiple values for each attribute added to a pattern (although some attributes are set and can only have one value). The pattern attribute matches a message attribute if any of the values in the pattern match the value in the message. If no value is specified for an attribute, the ToolTalk service assumes that any value will match.

RETURN VALUE

Upon successful completion, the *tt_pattern_create*( ) function returns the opaque handle for a message pattern. The application can use this handle in future calls to identify the pattern object. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_register*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_destroy — destroy a pattern object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_destroy(Tt_pattern p);
```

DESCRIPTION

The *tt_pattern_destroy*( ) function destroys a pattern object.

Destroying a pattern object automatically unregisters the pattern with the ToolTalk service.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called.

RETURN VALUE

Upon successful completion, the *tt_pattern_destroy*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_register*( ), *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_disposition_add — add a value to the disposition field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_disposition_add(Tt_pattern p,
                                     Tt_disposition r);
```

DESCRIPTION

The *tt_pattern_disposition_add*( ) function adds a value to the disposition field for the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called.

The *r* argument indicates whether an instance of the receiver is to be started to receive the message immediately, or whether the message is to be queued until the receiving process is started at a later time or discarded if the receiver is not started. The following values are defined:

TT_DISCARD
: There is no receiver for this message. The message will be returned to the sender with the **Tt_status** field containing TT_FAILED.

TT_QUEUE
: Queue the message until a process of the proper ptype receives the message.

TT_START
: Attempt to start a process of the proper ptype if none is running.

TT_QUEUE+TT_START
: Queue the message and attempt to start a process of the proper ptype if none is running.

RETURN VALUE

Upon successful completion, the *tt_pattern_disposition_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_file_add — add a value to the file field of a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_file_add(Tt_pattern p,
                              const char *file);
```

DESCRIPTION

The *tt_pattern_file_add*( ) function adds a value to the file field of the specified pattern.

The application can use this call to set individual files on individual patterns.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *file* argument is the name of the file of the specified pattern.

RETURN VALUE

Upon successful completion, the *tt_pattern_file_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

However, this call does not cause the pattern's ToolTalk session to be stored in the database.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_iarg_add — add a new integer argument to a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_iarg_add(Tt_pattern m,
                              Tt_mode n,
                              const char *vtype,
                              int value);
```

DESCRIPTION

The *tt_pattern_iarg_add*( ) function adds a new argument to a pattern and sets the value to a given integer.

Add all arguments before the pattern is registered with the ToolTalk service.

The *m* argument is the opaque handle for the pattern involved in this operation. The *n* argument specifies who (sender, handler, observers) writes and reads a message argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The *vtype* argument describes the type of argument data being added. NULL matches any value. The *value* argument is the value to be added.

RETURN VALUE

Upon successful completion, the *tt_pattern_iarg_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_MODE
: The specified **Tt_mode** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_register*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_icontext_add — add an integer value to the values of this pattern's context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_icontext_add(Tt_pattern p,
                                  const char *slotname,
                                  int value);
```

DESCRIPTION

The *tt_pattern_icontext_add*( ) function adds an integer value to the values of this pattern's context.

The *p* argument is the opaque handle for the pattern involved in this operation. The *slotname* argument describes the slotname in this pattern. The *value* argument is the value to be added.

RETURN VALUE

Tt_status Upon successful completion, the *tt_pattern_icontext_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_UNIMP
: The ToolTalk function called is not implemented.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_object_add — add a value to the object field of a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_object_add(Tt_pattern p,
                                const char *objid);
```

DESCRIPTION

The *tt_pattern_object_add*( ) function adds a value to the object field of the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *objid* argument is the identifier for the specified object. Both *tt_spec_create*( ) and *tt_spec_move*( ) return objids.

RETURN VALUE

Upon successful completion, the *tt_pattern_object_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_op_add — add a value to the operation field of a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_op_add(Tt_pattern p,
                            const char *opname);
```

DESCRIPTION

The *tt_pattern_op_add*( ) function adds a value to the operation field of the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *opname* argument is the name of the operation the process can perform.

RETURN VALUE

Upon successful completion, the *tt_pattern_op_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_opnum_add — add an operation number to a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_opnum_add(Tt_pattern p,
                               int opnum);
```

DESCRIPTION

The *tt_pattern_opnum_add*( ) function adds an operation number to the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *opnum* argument is the operation number to be added.

RETURN VALUE

Upon successful completion, the *tt_pattern_opnum_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_otype_add — add a value to the object type field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_otype_add(Tt_pattern p,
                               const char *otype);
```

DESCRIPTION

The *tt_pattern_otype_add*( ) function adds a value to the object type field for the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *otype* argument is the name of the object type the application manages.

RETURN VALUE

Upon successful completion, the *tt_pattern_otype_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_print — format a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_pattern_print(Tt_pattern *p);
```

DESCRIPTION

The *tt_pattern_print*( ) function formats a pattern in the same way a message is formatted for the *ttsession* trace and returns a string containing it.

The *p* argument is the pattern to be formatted.

RETURN VALUE

Upon successful completion, the *tt_pattern_print*( ) function returns the formatted string. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The *tt_pattern_print*( ) function allows an application writer to dump out patterns for debugging.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_register — register a pattern with the ToolTalk service

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_register(Tt_pattern p);
```

DESCRIPTION

The *tt_pattern_register*( ) function registers a pattern with the ToolTalk service.

When the process is registered, it will start receiving messages that match the specified pattern. Once a pattern is registered, no further changes can be made in the pattern.

When the process joins a session or file, the ToolTalk service updates the file and session field of its registered patterns.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called.

RETURN VALUE

Upon successful completion, the *tt_pattern_register*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_unregister*( ), *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_scope_add — add a value to the scope field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_scope_add(Tt_pattern p,
                               Tt_scope s);
```

DESCRIPTION

The *tt_pattern_scope_add*( ) function adds a value to the scope field for the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *s* argument specifies what processes are eligible to receive the message. The following values are defined:

TT_SESSION
: All processes joined to the indicated session are eligible.

TT_FILE
: All processes joined to the indicated file are eligible.

TT_BOTH
: All processes joined to either indicated file or the indicated session are eligible.

TT_FILE_IN_SESSION
: All processes joined to both the indicated file and the indicated session are eligible.

RETURN VALUE

Upon successful completion, the *tt_pattern_scope_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_sender_add — add a value to the sender field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_sender_add(Tt_pattern p,
                                const char *procid);
```

DESCRIPTION

The *tt_pattern_sender_add*( ) function adds a value to the sender field for the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *procid* argument is the character value that uniquely identifies the process of interest.

RETURN VALUE

Upon successful completion, the *tt_pattern_sender_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_sender_ptype_add — add a value to the sending process's ptype field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_sender_ptype_add(Tt_pattern p,
                                      const char *ptid);
```

DESCRIPTION

The *tt_pattern_sender_ptype_add*( ) function adds a value to the sending process's ptype field for the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *ptid* argument is the character string that uniquely identifies the type of process in which the application is interested.

RETURN VALUE

Upon successful completion, the *tt_pattern_sender_ptype_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_session_add — adds a value to the session field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_session_add(Tt_pattern p,
                                 const char *sessid);
```

DESCRIPTION

The *tt_pattern_session_add*( ) function adds a value to the session field for the specified pattern.

When the process joins a session, the ToolTalk service updates the session field of its registered patterns.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *sessid* argument is the session of interest.

RETURN VALUE

Upon successful completion, the *tt_pattern_session_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_state_add — add a value to the state field for a pattern

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_state_add(Tt_pattern p,
                               Tt_state s);
```

DESCRIPTION

The *tt_pattern_state_add*( ) function adds a value to the state field for the specified pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *s* argument indicates the current delivery state of a message. The following values are defined:

TT_CREATED
: The message has been created, but not yet sent.

TT_SENT
: The message has been sent, but not yet handled.

TT_HANDLED
: The message has been handled; return values are valid.

TT_FAILED
: The message could not be delivered to a handler.

TT_QUEUED
: The message has been queued for delivery.

TT_STARTED
: The ToolTalk service is attempting to start a process to handle the message.

TT_REJECTED
: The message has been rejected by a possible handler.

RETURN VALUE

Upon successful completion, the *tt_pattern_state_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_unregister — unregister a pattern from the ToolTalk service

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_unregister(Tt_pattern p);
```

DESCRIPTION

The *tt_pattern_unregister*( ) function unregisters the specified pattern from the ToolTalk service. The process will stop receiving messages that match this pattern.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called.

RETURN VALUE

Upon successful completion, the *tt_pattern_unregister*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_register*( ), *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_user — return the value in a user data cell for a pattern object

SYNOPSIS

```
#include <Tt/tt_c.h>

void *tt_pattern_user(Tt_pattern p,
                      int key);
```

DESCRIPTION

The *tt_pattern_user*( ) function returns the value in the indicated user data cell for the specified pattern object.

Every pattern object allows an arbitrary number of user data cells that are each one word in size. The user data cells are identified by integer keys. The tool can use these keys in any manner to associate arbitrary data with a pattern object.

The user data is part of the pattern object (that is, the storage buffer in the application); it is not part of the actual pattern. The content of user cells has no effect on pattern matching.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *key* argument is the specified user data cell. The application can use *tt_pattern_user_set*( ) to assign the keys to the user data cells that are part of the pattern object. The value of each data cell must be unique for this pattern.

RETURN VALUE

Upon successful completion, the *tt_pattern_user*( ) function returns the data cell, a piece of arbitrary user data that can hold a **void** *. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned data:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

APPLICATION USAGE

The user data cell is intended to hold an address. If the address selected is equal to one of the **Tt_status** enumerated values, the result of the *tt_ptr_error*( ) function will not be reliable.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_user_set*( ), *tt_pattern_create*( ), *tt_ptr_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_user_set — store information in the user data cells of a pattern object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_user_set(Tt_pattern p,
                              int key,
                              void *v);
```

DESCRIPTION

The *tt_pattern_user_set*( ) function stores information in the user data cells associated with the specified pattern object.

The *p* argument is a unique handle for a message pattern. This handle is returned after *tt_pattern_create*( ) is called. The *key* argument is the specified user data cell. The value for each data cell must be unique for this pattern. The *v* argument is the data cell, a piece of arbitrary user data that can hold a **void** *.

RETURN VALUE

Upon successful completion, the *tt_pattern_user_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_pattern_user*( ), *tt_pattern_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_xarg_add — add a new argument with an interpreted XDR value to a pattern object

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_xarg_add(Tt_pattern m,
                              Tt_mode n,
                              const char *vtype,
                              xdrproc_t xdr_proc,
                              void *value);
```

DESCRIPTION

The *tt_pattern_xarg_add*( ) function adds a new argument with an interpreted XDR value to a pattern object.

The *m* argument is the opaque handle for the pattern involved in this operation. The *n* argument specifies who (sender, handler, observers) writes and reads a pattern argument. The following modes are defined:

TT_IN The argument is written by the sender and read by the handler and any observers.

TT_OUT
: The argument is written by the handler and read by the sender and any reply observers.

TT_INOUT
: The argument is written by the sender and the handler and read by all.

The *vtype* argument describes the type of argument data being added. The *xdr_proc* argument points to the XDR procedure to be used to serialise the data pointed to by *value*. The *value* argument is the data to be serialised.

RETURN VALUE

Upon successful completion, the *tt_pattern_xarg_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_MODE
: The specified **Tt_mode** is invalid.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length expression.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pattern_xcontext_add — add an XDR-interpreted byte-array value to this pattern's named context

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pattern_xcontext_add(Tt_pattern p,
                                  const char *slotname,
                                  xdrproc_t xdr_proc,
                                  void *value);
```

DESCRIPTION

The *tt_pattern_xcontext_add*( ) function adds an XDR-interpreted byte-array value to the values in this pattern's named context.

The *p* argument is the opaque handle for the pattern involved in this operation. The *slotname* argument describes the context for this pattern. The *xdr_proc* argument points to the XDR procedure to be used to serialise the data pointed to by value. The *value* argument is the data to be serialised.

RETURN VALUE

Upon successful completion, the *tt_pattern_xcontext_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_UNIMP
: The ToolTalk function called is not implemented.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length expression.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pnotice_create — create a procedure notice

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_message tt_pnotice_create(Tt_scope scope,
                             const char *op);
```

DESCRIPTION

The *tt_pnotice_create*( ) function creates a message. The created message contains the following:

Tt_address = TT_PROCEDURE
Tt_class = TT_NOTICE

The application can use the returned handle to add arguments and other attributes, and to send the message.

The *scope* argument determines which processes are eligible to receive the message. The following values are defined:

TT_SESSION
: All processes joined to the indicated session are eligible.

TT_FILE
: All processes joined to the indicated file are eligible.

TT_BOTH
: All processes joined to either indicated file or the indicated session are eligible.

TT_FILE_IN_SESSION
: All processes joined to both the indicated file and the indicated session are eligible.

The *op* argument is the operation to be performed by the receiving process.

RETURN VALUE

Upon successful completion, the *tt_pnotice_create*( ) function returns the unique handle that identifies this message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

If the ToolTalk service is unable to create a message when requested, *tt_pnotice_create*( ) returns an invalid handle. When the application attempts to use this handle with another ToolTalk function, the ToolTalk service will return TT_ERR_POINTER.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_pointer_error — return the status of a pointer

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_pointer_error(void *pointer);
```

DESCRIPTION

The *tt_pointer_error*( ) function returns the status of the specified pointer.

If an opaque pointer (**Tt_message** or **Tt_pattern**) or character pointer **(char *)** is specified, this function returns TT_OK if the pointer is valid or the encoded **Tt_status** value if the pointer is an error object.

The *pointer* argument is the opaque pointer or character pointer to be checked.

RETURN VALUE

Upon successful completion, the *tt_pointer_error*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_ptr_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_prequest_create — create a procedure request message

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_message tt_prequest_create(Tt_scope scope,
                              const char *op);
```

DESCRIPTION

The *tt_prequest_create*( ) function creates a message. The created message created contains the following:

Tt_address = TT_PROCEDURE
Tt_class = TT_REQUEST

The application can use the returned handle to add arguments and other attributes, and to send the message.

The *scope* argument determines which processes are eligible to receive the message. The following values are defined:

TT_SESSION
: All processes joined to the indicated session are eligible.

TT_FILE
: All processes joined to the indicated file are eligible.

TT_BOTH
: All processes joined to either indicated file or the indicated session are eligible.

TT_FILE_IN_SESSION
: All processes joined to both the indicated file and the indicated session are eligible.

The *op* argument is the operation to be performed by the receiving process.

RETURN VALUE

Upon successful completion, the *tt_prequest_create*( ) function returns the unique handle that identifies this message. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

If the ToolTalk service is unable to create a message when requested, *tt_prequest_create*( ) returns an invalid handle. When the application attempts to use this handle with another ToolTalk function, the ToolTalk service will return TT_ERR_POINTER.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_ptr_error — pointer error macro

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_ptr_error(pointer);
```

DESCRIPTION

The *tt_ptr_error*( ) macro expands to **tt_pointer_error((void *)(p))**.

The *pointer* argument is the opaque pointer or character pointer to be checked.

RETURN VALUE

Upon successful completion, the *tt_ptr_error*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_ptype_declare — register the process type with the ToolTalk service

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_ptype_declare(const char *ptid);
```

DESCRIPTION

The *tt_ptype_declare*( ) function registers the process type with the ToolTalk service.

The *ptid* argument is the character string specified in the ptype that uniquely identifies this process.

RETURN VALUE

Upon successful completion, the *tt_ptype_declare*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PTYPE
: The specified process type is not the name of an installed process type.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_ptype_exists — indicate whether a ptype is already installed

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_ptype_exists(const char *ptid);
```

DESCRIPTION

The *tt_ptype_exists*( ) function returns an indication of whether a ptype is already installed.

The *ptid* argument is the character string specifying the ptype.

RETURN VALUE

Upon successful completion, the *tt_ptype_exists*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully and the ptype is already installed.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PTYPE
: The specified process type is not the name of an installed process type.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_ptype_opnum_callback_add — return a callback if two opnums are equal

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_ptype_opnum_callback_add(const char *ptid,
                                      int opnum,
                                      Tt_message_callback f);
```

DESCRIPTION

The *tt_ptype_opnum_callback_add* ( ) function returns a callback if the specified opnums are equal. The callback is defined in **<Tt/tt_c.h>**.

When a message is delivered because it matched a pattern derived from a signature in the named ptype with an opnum equal to the specified one, the given callback is run in the usual ToolTalk way.

The *ptid* argument is the identifier of the ptype involved in this operation. The *opnum* argument is the opnum of the specified ptype. The *f* argument is the message callback to be run.

RETURN VALUE

Upon successful completion, the *tt_ptype_opnum_callback_add* ( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_PTYPE
: The specified process type is not the name of an installed process type.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

APPLICATION USAGE

The *tt_ptype_opnum_callback_add* ( ) function will only be called for messages delivered by virtue of matching handler signatures. The callback cannot be called for observer signatures because the observer ptype is not recorded in the incoming message.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_ptype_undeclare — undeclare a ptype

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_ptype_undeclare(const char *ptid);
```

DESCRIPTION

The *tt_ptype_undeclare*( ) function undeclares the indicated ptype and unregisters the patterns associated with the indicated ptype from the ToolTalk service.

The *ptid* argument is the character string specifying the ptype.

RETURN VALUE

Upon successful completion, the *tt_ptype_undeclare*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PTYPE
: The specified process type is not the name of an installed process type.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_release — free storage allocated on the ToolTalk API allocation stack

SYNOPSIS

```
#include <Tt/tt_c.h>

void tt_release(int mark);
```

DESCRIPTION

The *tt_release*( ) function frees all storage allocated on the ToolTalk API allocation stack since *mark* was returned by *tt_mark*( ).

The *mark* argument is an integer that marks the application's storage position in the ToolTalk API allocation stack.

APPLICATION USAGE

This function frees all storage allocated since the *tt_mark*( ) call that returned *mark* and is typically called at the end of a procedure to release all storage allocated within the procedure.

SEE ALSO

<Tt/tt_c.h>, *tt_mark*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_bprop — retrieve the ith value of the named property of a session

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_session_bprop(const char *sessid,
                           const char *propname,
                           int i,
                           unsigned char **value,
                           int *length);
```

DESCRIPTION

The *tt_session_bprop*( ) function retrieves the *i*th value of the named property of the specified session.

If there are *i* values or fewer, both the returned value and the returned length are set to zero.

The *sessid* argument is the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called. The *propname* argument is the name of the property from which values are to be obtained. The *i* argument is the number of the item in the property list from which the value is to be obtained. The list numbering begins with zero. The *value* argument is the address of a character pointer to which the ToolTalk service is to point a string that contains the contents of the property. The *len* argument is the address of an integer to which the ToolTalk service is to set the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_session_bprop*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_bprop_add — add a new byte-string value to the end of the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_session_bprop_add(const char *sessid,
                               const char *propname,
                               const unsigned char *value,
                               int length);
```

DESCRIPTION

The *tt_session_bprop_add*( ) function adds a new byte-string value to the end of the list of values for the named property of the specified session.

The *sessid* argument is the name of the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called. The *propname* argument is the name of the property to which to add values. The *value* argument is the value to add to the session property. The *length* argument is the size of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_session_bprop_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROPLEN
: The specified property value is too long. (The maximum size is implementation specific, but is at least 2048.)

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_bprop_set — replace current values stored under the named property of a session

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_session_bprop_set(const char *sessid,
                               const char *propname,
                               const unsigned char *value,
                               int length);
```

DESCRIPTION

The *tt_session_bprop_set*( ) function replaces any current values stored under the named property of the specified session with the given byte-string value.

The *sessid* argument is the name of the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called. The *propname* argument is the name of the property whose value is to be replaced. The *value* argument is the value to which the session property is set. If value is NULL, the property is removed entirely. The *length* argument is the size of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_session_bprop_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROPLEN
: The specified property value is too long. (The maximum size is implementation specific, but is at least 2048.)

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_join — join a session and make it the default

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_session_join(const char *sessid);
```

DESCRIPTION

The *tt_session_join*( ) function joins the named session and makes it the default session.

The *sessid* argument is the name of the session to join.

RETURN VALUE

Upon successful completion, the *tt_session_join*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

APPLICATION USAGE

The application can use the *sessid* value returned by *tt_default_session*( ), *tt_X_session*( ), or *tt_initial_session*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_X_session*( ), *tt_default_session*( ), *tt_initial_session*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_prop — return the ith value of a session property

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_session_prop(const char *sessid,
                      const char *propname,
                      int i);
```

DESCRIPTION

The *tt_session_prop*( ) function returns the *i*th value of the specified session property.

The *sessid* argument is the name of the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called. The *propname* argument is the name of the property from which a value is to be retrieved. The name must be less than 64 bytes. The *i* argument is the number of the item in the property name list for which the value is to be obtained. The list numbering begins with zero.

RETURN VALUE

Upon successful completion, the *tt_session_prop*( ) function returns the value of the requested property. If there are *i* values or fewer, it returns NULL. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

If the returned value has embedded nulls, it is impossible to determine how long it is. The application can use *tt_session_bprop*( ) for values with embedded nulls.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_prop_add — add a new character-string value to the end of the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_session_prop_add(const char *sessid,
                              const char *propname,
                              const char *value);
```

DESCRIPTION

The *tt_session_prop_add*( ) function adds a new character-string value to the end of the list of values for the property of the specified session.

The *sessid* argument is the name of the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called. The *propname* argument is the name of the property to which a value is to be added. The name must be less than 64 bytes. The *value* argument is the character string to add to the property name list.

RETURN VALUE

Upon successful completion, the *tt_session_prop_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROPLEN
: The specified property value is too long. (The maximum size is implementation specific, but is at least 64.)

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_prop_count — return the number of values stored under a property of a session

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_session_prop_count(const char *sessid,
                          const char *propname);
```

DESCRIPTION

The *tt_session_prop_count*( ) function returns the number of values stored under the named property of the specified session.

The *sessid* argument is the name of the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called. The *propname* argument is the name of the property to be examined.

RETURN VALUE

Upon successful completion, the *tt_session_prop_count*( ) function returns the number of values in the specified property list. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_prop_set — replace current values for a property of a session with a character-string value

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_session_prop_set(const char *sessid,
                              const char *propname,
                              const char *value);
```

DESCRIPTION

The *tt_session_prop_set*( ) function replaces all current values stored under the named property of the specified session with the given character-string value.

The *sessid* argument is the name of the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called. The *propname* argument is the name of the property to be examined. The *value* argument is the new value to be inserted. NULL removes a value from the property list.

RETURN VALUE

Upon successful completion, the *tt_session_prop_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROPLEN
: The specified property value is too long. (The maximum size is implementation specific, but is at least 64.)

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_propname — returns an element of the list of property names for a session

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_session_propname(const char *sessid,
                          int n);
```

DESCRIPTION

The *tt_session_propname*( ) function returns the *n*th element of the list of currently defined property names for the specified session.

The *sessid* argument is the name of the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called. The *n* argument is the number of the item in the property name list for which a name is to be obtained. The list numbering begins with zero.

RETURN VALUE

Upon successful completion, the *tt_session_propname*( ) function returns the name of the specified property from the session property list. If there are *n* properties or fewer, *tt_session_propname*( ) returns NULL. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_propnames_count — return the number of property names for the session

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_session_propnames_count(const char *sessid);
```

DESCRIPTION

The *tt_session_propnames_count*( ) function returns the number of currently defined property names for the session.

The *sessid* argument is the name of the session joined. The application can use the *sessid* value returned when *tt_default_session*( ) is called.

RETURN VALUE

Upon successful completion, the *tt_session_propnames_count*( ) function returns the number of property names for the session. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_quit — quit the session

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_session_quit(const char *sessid);
```

DESCRIPTION

The *tt_session_quit*( ) function informs the ToolTalk service that the process is no longer interested in this ToolTalk session. The ToolTalk service stops delivering messages scoped to this session.

The *sessid* argument is the name of the session to quit.

RETURN VALUE

Upon successful completion, the *tt_session_quit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_session_types_load — merge a compiled ToolTalk types file into the running ttsession

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_session_types_load(const char *session,
                                const char *filename);
```

DESCRIPTION

The *tt_session_types_load*( ) function merges a compiled ToolTalk types file into the running *ttsession*.

The *session* argument is the name of the running session. The *filename* argument is the name of the compiled ToolTalk types file.

RETURN VALUE

Upon successful completion, the *tt_session_types_load*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

TT_ERR_FILE
: The specified file does not exist or it is inaccessible.

TT_ERR_UNIMP
: The ToolTalk function called is not implemented.

SEE ALSO

<Tt/tt_c.h>, *ttsession*.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_bprop — retrieve the ith value of a property

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_spec_bprop(const char *objid,
                        const char *propname,
                        int i,
                        unsigned char **value,
                        int *length);
```

DESCRIPTION

The *tt_spec_bprop*( ) function retrieves the *i*th value of the specified property.

The *objid* argument is the identifier of the object involved in this operation. The *propname* argument is the name of the property whose value is to be retrieved. The name must be less than 64 characters. The *i* argument is the item of the list for which a value is to be obtained. The list numbering begins with zero. The *value* argument is the address of a character pointer to which the ToolTalk service is to point a string that contains the contents of the spec's property. If there are *i* values or fewer, the pointer is set to zero. The *length* argument is the address of an integer to which the ToolTalk service is to set the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_spec_bprop*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_bprop_add — add a new byte-string to the end of the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_spec_bprop_add(const char *objid,
                            const char *propname,
                            const unsigned char *value,
                            int length);
```

DESCRIPTION

The *tt_spec_bprop_add*( ) function adds a new byte-string to the end of the list of values associated with the specified spec property.

The *objid* argument is the identifier of the object involved in this operation. The *propname* argument is the name of the property to which the byte-string is to be added. The *value* argument is the byte-string to be added to the property value list. The *length* argument is the length in bytes of the byte-string.

RETURN VALUE

Upon successful completion, the *tt_spec_bprop_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_PROPLEN
: The specified property value is too long. (The maximum size is implementation specific, but is at least 64.)

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_bprop_set — replace any current values stored under this spec property with a new byte-string

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_spec_bprop_set(const char *objid,
                            const char *propname,
                            const unsigned char *value,
                            int length);
```

DESCRIPTION

The *tt_spec_bprop_set*( ) function replaces any current values stored under this spec property with a new byte-string.

The *objid* argument is the identifier of the object involved in this operation. The *propname* argument is the name of the property which stores the values. The *value* argument is the byte-string to be added to the property value list. If the value is NULL, the property is removed entirely. The *length* argument is the length of the value in bytes.

RETURN VALUE

Upon successful completion, the *tt_spec_bprop_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_PROPLEN
: The specified property value is too long. (The maximum size is implementation specific, but is at least 64.)

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_create — create an in-memory spec for an object

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_spec_create(const char *filepath);
```

DESCRIPTION

The *tt_spec_create*( ) function creates a spec (in memory) for an object.

The application can use the objid returned in future calls to manipulate the object.

The *filepath* argument is the name of the file.

RETURN VALUE

Upon successful completion, the *tt_spec_create*( ) function returns the identifier for this object. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

TT_ERR_PATH
: The specified pathname included an unsearchable directory.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

To make the object a permanent ToolTalk item or one visible to other processes, the creating process must call *tt_spec_write*( ).

SEE ALSO

<Tt/tt_c.h>, *tt_spec_type_set*( ), *tt_spec_write*( ), *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_destroy — destroy an object's spec

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_spec_destroy(const char *objid);
```

DESCRIPTION

The *tt_spec_destroy*( ) function destroys an object's spec immediately.

The *objid* argument is the identifier of the object involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_spec_destroy*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

SEE ALSO

<Tt/tt_c.h>.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_file — retrieve the name of the file that contains the object described by the spec

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_spec_file(const char *objid);
```

DESCRIPTION

The *tt_spec_file*( ) function retrieves the name of the file that contains the object described by the spec.

The *objid* argument is the identifier of the object involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_spec_file*( ) function returns the absolute pathname of the file. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_move — notify the ToolTalk service that an object has moved to a different file

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_spec_move(const char *objid,
                   const char *newfilepath);
```

DESCRIPTION

The *tt_spec_move*( ) function notifies the ToolTalk service that this object has moved to a different file.

The ToolTalk service returns a new objid for the object and leaves a forwarding pointer from the old objid to the new one.

If a new objid is not required (for example, because the new and old files are in the same file system), *tt_spec_move*( ) returns TT_WRN_SAME_OBJID.

The *objid* argument is the identifier of the object involved in this operation.

The *newfilepath* argument is the new file name.

RETURN VALUE

Upon successful completion, the *tt_spec_move*( ) function returns the new unique identifier of the object involved in this operation. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_PATH
: The specified pathname included an unsearchable directory.

TT_WRN_SAME_OBJID
: A new *objid* is not required.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

For efficiency and reliability, the application should replace any references in the application to the old *objid* with references to the new one.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_prop — retrieve the ith value of the property associated with an object spec

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_spec_prop(const char *objid,
                   const char *propname,
                   int i);
```

DESCRIPTION

The *tt_spec_prop*( ) function retrieves the *i*th value of the property associated with this object spec.

The *objid* argument is the identifier of the object involved in this operation. The *propname* argument is the name of the property associated with the object spec. The *i* argument is the item of the list whose value is to be retrieved. The list numbering begins with zero.

RETURN VALUE

Upon successful completion, the *tt_spec_prop*( ) function returns the contents of the property value. If there are *i* values or less, *tt_spec_prop*( ) returns NULL. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

If the returned value has embedded nulls, its length cannot be determined.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_prop_add — add a new item to the end of the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_spec_prop_add(const char *objid,
                           const char *propname,
                           const char *value);
```

DESCRIPTION

The *tt_spec_prop_add*( ) function adds a new item to the end of the list of values associated with this spec property.

The *objid* argument is the identifier of the object involved in this operation. The *propname* argument is the property to which the item is to be added. The *value* argument is the new character-string to be added to the property value list.

RETURN VALUE

Upon successful completion, the *tt_spec_prop_add*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_PROPLEN
: The specified property value is too long. (The maximum size is implementation specific, but is at least 2048.)

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>, *tt_spec_prop_set*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_prop_count — return the number of values listed in this spec property

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_spec_prop_count(const char *objid,
                       const char *propname);
```

DESCRIPTION

The *tt_spec_prop_count*( ) function returns the number of values listed in this spec property.

The *objid* argument is the identifier of the object involved in this operation. The *propname* argument is the name of the property that contains the value to be returned.

RETURN VALUE

Upon successful completion, the *tt_spec_prop_count*( ) function returns the number of values listed in the spec property. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_prop_set — replace property values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_spec_prop_set(const char *objid,
                           const char *propname,
                           const char *value);
```

DESCRIPTION

The *tt_spec_prop_set*( ) function replaces any values currently stored under this property of the object spec with a new value.

The *objid* argument is the identifier of the object involved in this operation. The *propname* argument is the name of the property which stores the values. The *value* argument is the value to be placed in the property value list. If value is NULL, the property is removed entirely.

RETURN VALUE

Upon successful completion, the *tt_spec_prop_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_PROPLEN
: The specified property value is too long. (The maximum size is implementation specific, but is at least 2048.)

TT_ERR_PROPNAME
: The specified property name is syntactically invalid.

SEE ALSO

<Tt/tt_c.h>, *tt_spec_prop_add*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_propname — return an element of the property name list for an object spec

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_spec_propname(const char *objid,
                       int n);
```

DESCRIPTION

The *tt_spec_propname*( ) function returns the *n*th element of the property name list for this object spec.

The *objid* argument is the identifier of the object involved in this operation. The *n* argument is the item of the list whose element is to be returned. The list numbering begins with zero.

RETURN VALUE

Upon successful completion, the *tt_spec_propname*( ) function returns the property name. If there are *n* properties or less, *tt_spec_propname*( ) returns NULL. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_propnames_count — return the number of property names for an object

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_spec_propnames_count(const char *objid);
```

DESCRIPTION

The *tt_spec_propnames_count*( ) function returns the number of property names for this object.

The *objid* argument is the identifier of the object involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_spec_propnames_count*( ) function returns the number of values listed in the spec property. The application can use *tt_int_error*( ) to extract one of the following **Tt_status** values from the returned integer:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

SEE ALSO

<Tt/tt_c.h>, *tt_int_error*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_type — return the name of the object type

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_spec_type(const char *objid);
```

DESCRIPTION

The *tt_spec_type*( ) function returns the name of the object type.

The *objid* argument is the identifier of the object involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_spec_type*( ) function returns the type of this object. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned pointer:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_type_set — assign an object type value to an object spec

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_spec_type_set(const char *objid,
                           const char *otid);
```

DESCRIPTION

The *tt_spec_type_set*( ) function assigns an object type value to the object spec.

The type must be set before the spec is written for the first time and cannot be set thereafter.

The *objid* argument is the identifier of the object involved in this operation. The *otid* argument is the otype to be assigned to the spec.

RETURN VALUE

Upon successful completion, the *tt_spec_type_set*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_READONLY
: The attribute the application is trying to change is not owned or writable by the current user.

SEE ALSO

<Tt/tt_c.h>, *tt_spec_create*( ), *tt_spec_write*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_spec_write — write the spec and any associated properties to the ToolTalk database

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_spec_write(const char *objid);
```

DESCRIPTION

The *tt_spec_write*( ) function writes the spec and any associated properties to the ToolTalk database. The type must be set before the spec is written for the first time.

The *objid* argument is the identifier of the object involved in this operation.

RETURN VALUE

Upon successful completion, the *tt_spec_write*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OBJID
: The *objid* passed to the ToolTalk service does not reference an existing object spec.

TT_ERR_OTYPE
: The specified object type is not the name of an installed object type.

APPLICATION USAGE

It is not necessary to perform a write operation after a destroy operation.

Several changes can be batched between write calls; for example, the application can create an object spec, set some properties, and then write all the changes at once with one write call.

SEE ALSO

<Tt/tt_c.h>, *tt_spec_create*( ), *tt_spec_type_set*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_status_message — provide a message for a problem status code

SYNOPSIS

```
#include <Tt/tt_c.h>

char *tt_status_message(Tt_status ttrc);
```

DESCRIPTION

The *tt_status_message*( ) function returns a pointer to a message that describes the problem indicated by this status code.

The *ttrc* argument is the status code received during an operation.

RETURN VALUE

Upon successful completion, the *tt_status_message*( ) function returns a pointer to a character string that describes the status code, which is one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_xxx Any other TT_ status code is explained in the returned string.

APPLICATION USAGE

The application should use *tt_free*( ) to free any data stored in the address returned by the ToolTalk API.

SEE ALSO

<Tt/tt_c.h>, *tt_ptr_error*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tt_trace_control — control client-side tracing

SYNOPSIS

```
#include <Tt/tt_c.h>

int tt_trace_control(int onoff);
```

DESCRIPTION

The *tt_trace_control*( ) function sets or clears an internal flag controlling all client-side tracing. The intent of this is to be called from debugger breakpoints, allowing a programmer to narrow the trace to the suspect area.

The value of the *onoff* argument affects tracing as follows:

0 Tracing is turned off.

1 Tracing is turned on.

–1 Tracing is turned on if it was off and vice-versa.

RETURN VALUE

The *tt_trace_control*( ) function returns the previous setting of the trace flag.

APPLICATION USAGE

This call does not return one of the TT_xxx type of errors or warnings, but only the numbers 1 or zero.

SEE ALSO

<Tt/tt_c.h>, *ttsession*.

CHANGE HISTORY

First released in Issue 1.

NAME

tt_xcontext_quit — remove an XDR-interpreted byte-array value from the list of values

SYNOPSIS

```
#include <Tt/tt_c.h>

Tt_status tt_xcontext_quit(const char *slotname,
                           xdrproc_t xdr_proc,
                           void *value);
```

DESCRIPTION

The *tt_xcontext_quit*( ) function removes the given XDR-interpreted byte-array value from the list of values for the contexts of all patterns.

The *slotname* argument describes the slotname in this message. The *xdr_proc* argument points to the XDR procedure to be used to serialise the data pointed to by *value*.

The *value* argument is the data to be serialised.

RETURN VALUE

Upon successful completion, the *tt_xcontext_quit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_SLOTNAME
: The specified slotname is syntactically invalid.

TT_ERR_XDR
: The XDR procedure failed on the given data, or evaluated to a zero-length expression.

SEE ALSO

<Tt/tt_c.h>; the referenced **XDR** specification.

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_Get_Modified — ask if any ToolTalk client has changes pending on a file

SYNOPSIS

```
#include <Tt/tttk.h>

int ttdt_Get_Modified(Tt_message context,
                      const char *pathname,
                      Tt_scope the_scope,
                      XtAppContext app2run,
                      int ms_timeout);
```

DESCRIPTION

The *ttdt_Get_Modified*( ) function sends a *Get_Modified* request in the scope *the_scope* and waits for the reply. A *Get_Modified* request asks if any ToolTalk client has changes pending on *pathname* that it intends to make persistent.

The *context* argument describes the environment to use. If *context* is not zero, messages created by *ttdt_Get_Modified*( ) inherit from *context* all contexts whose slotname begins with the characters ENV_. That is, the environment described in *context* is propagated to messages created by *ttdt_Get_Modified*( ).

The *pathname* argument is a pointer to a pathname on which the client is operating.

The *the_scope* argument identifies the scope of the request. If *the_scope* is TT_SCOPE_NONE, *ttdt_Get_Modified*( ) tries TT_BOTH, and falls back to TT_FILE_IN_SESSION if, for example, the ToolTalk database server is not installed on the file server that owns *pathname*.

The *ttdt_Get_Modified*( ) function passes *app2run* and *ms_timeout* to *tttk_block_while*( ), blocking on the reply to the *Get_Modified* request it sends.

RETURN VALUE

Upon successful completion, the *ttdt_Get_Modified*( ) function returns non-zero if the *Get_Modified* request receives an affirmative reply within *ms_timeout* milliseconds; otherwise, it returns zero.

SEE ALSO

<Tt/tttk.h>, *ttdt_file_join*( ), *ttdt_file_event*( ), *tttk_block_while*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_Revert — request a ToolTalk client to revert a file

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttdt_Revert(Tt_message context,
                      const char *pathname,
                      Tt_scope the_scope,
                      XtAppContext app2run,
                      int ms_timeout);
```

DESCRIPTION

The *ttdt_Revert*( ) function sends a *Revert* request in the *the_scope* argument and waits for the reply. A *Revert* request asks the handling ToolTalk client to discard any changes pending on *pathname*.

The *context* argument describes the environment to use. If *context* is not zero, messages created by *ttdt_Revert*( ) inherit from *context* all contexts whose slotname begins with the characters ENV_.

The *the_scope* argument identifies the scope of the request. If *the_scope* is TT_SCOPE_NONE, *ttdt_Revert*( ) tries TT_BOTH, and falls back to TT_FILE_IN_SESSION if, for example, the ToolTalk database server is not installed on the file server that owns *pathname*.

The *ttdt_Revert*( ) function passes *app2run* and *ms_timeout* to *tttk_block_while*( ), blocking on the reply to the *Save* request it sends.

RETURN VALUE

Upon successful completion, the *ttdt_Revert*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The sent request received an affirmative reply within *ms_timeout* milliseconds.

TT_DESKTOP_ETIMEDOUT
: No reply was received within *ms_timeout* milliseconds.

TT_DESKTOP_EPROTO
: The request was failed, but the handler set the *tt_message_status*( ) of the failure reply to TT_OK, instead of a specific error status.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_POINTER
: The *pathname* argument was NULL or was a ToolTalk error pointer.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tttk.h>, *ttdt_Save*( ), *ttdt_file_join*( ), *ttdt_file_event*( ), *tttk_block_while*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_Save — request a ToolTalk client to save a file

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttdt_Save(Tt_message context,
                    const char *pathname,
                    Tt_scope the_scope,
                    XtAppContext app2run,
                    int ms_timeout);
```

DESCRIPTION

The *ttdt_Save*( ) function sends a *Save* request in the *the_scope* argument and waits for the reply. A *Save* request asks the handling ToolTalk client to save any changes pending on *pathname*.

The *context* argument describes the environment to use. If *context* is not zero, messages created by *ttdt_Save*( ) inherit from *context* all contexts whose slotname begins with the characters ENV_.

The *the_scope* argument identifies the scope of the request. If *the_scope* is TT_SCOPE_NONE, *ttdt_Save*( ) tries TT_BOTH, and falls back to TT_FILE_IN_SESSION if, for example, the ToolTalk database server is not installed on the file server that owns *pathname*.

The *ttdt_Save*( ) function passes *app2run* and *ms_timeout* to *tttk_block_while*( ), blocking on the reply to the *Save* request it sends.

RETURN VALUE

Upon successful completion, the *ttdt_Save*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The sent request received an affirmative reply within *ms_timeout* milliseconds.

TT_DESKTOP_ETIMEDOUT
: No reply was received within *ms_timeout* milliseconds.

TT_DESKTOP_EPROTO
: The request was failed, but the handler set the *tt_message_status*( ) of the failure reply to TT_OK, instead of a specific error status.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_POINTER
: The *pathname* argument was NULL or was a ToolTalk error pointer.

TT_ERR_PROCID
The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tttk.h>, *ttdt_Revert*( ), *ttdt_file_join*( ), *ttdt_file_event*( ), *tttk_block_while*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_close — destroy a ToolTalk communication endpoint

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttdt_close(const char *procid,
                     const char *new_procid,
                     int sendStopped);
```

DESCRIPTION

The *ttdt_close*( ) function destroys a ToolTalk communication endpoint.

If *sendStopped* is True, the *ttdt_close*( ) function sends a *Stopped* notice; otherwise, it sends no notice. If *procid* is not NULL, *ttdt_close*( ) calls *tt_default_procid_set*( ) with a *procid* argument and then calls *tt_close*( ); otherwise, it closes the current default procid. If *new_procid* is not NULL, *ttdt_close*( ) calls *tt_default_procid_set*( ) with a *new_procid* argument.

RETURN VALUE

The *ttdt_close*( ) function may return any of the errors returned by *tt_default_procid_set*( ) and *tt_close*( ).

No errors are propagated if sending the *Stopped* notice fails.

SEE ALSO

<Tt/tttk.h>, *ttdt_open*( ), *tt_default_procid_set*( ), *tt_close*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_file_event — use ToolTalk to announce an event about a file

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttdt_file_event(Tt_message context,
                          Tttk_op event,
                          Tt_pattern *patterns,
                          int send);
```

DESCRIPTION

The *ttdt_file_event*( ) function is used to create and send a ToolTalk notice announcing an event pertaining to a file. The file is indicated by the *pathname* argument that was passed to *ttdt_file_join*( ) when *patterns* was created.

The *event* argument identifies the event. If *event* is TTDT_MODIFIED, *ttdt_file_event*( ) registers in the *the_scope* argument passed to *ttdt_file_join*( ) to handle *Get_Modified*, *Save*, and *Revert* requests. *Get_Modified* is handled transparently by associating the modified state of the file with *patterns*. *Save* and *Revert* requests are passed to the **Ttdt_file_cb** that was given to *ttdt_file_join*( ). If *send* is True, *ttdt_file_event*( ) sends *Modified* in *the_scope*. If *event* is TTDT_SAVED or TTDT_REVERTED, *ttdt_file_event*( ) unregisters handler patterns for *Get_Modified*, *Save*, and *Revert* requests. If *send* is True, *ttdt_file_event*( ) sends *Saved* or *Reverted*, respectively, in *the_scope*.

The *context* argument describes the environment to use. If *context* is not zero, messages created by *ttdt_file_event*( ) inherit from *context* all contexts whose slotname begins with the characters ENV_.

RETURN VALUE

Upon successful completion, the *ttdt_file_event*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_POINTER
: The *patterns* argument was NULL.

SEE ALSO

<Tt/tttk.h>, *ttdt_file_join*( ), *ttdt_Get_Modified*( ), *ttdt_file_quit*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_file_join — register to observe ToolTalk events on a file

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_pattern *ttdt_file_join(const char *pathname,
                           Tt_scope the_scope,
                           int join,
                           Ttdt_file_cb cb,
                           void *clientdata);
```

DESCRIPTION

The *ttdt_file_join*( ) function registers to observe *Deleted*, *Modified*, *Reverted*, *Moved*, and *Saved* notices.

If *join* is True, *ttdt_file_join*( ) calls *tt_file_join*( ) with a *pathname* argument.

The *the_scope* argument identifies the scope of the request. If *the_scope* is TT_SCOPE_NONE, it tries TT_BOTH, and falls back to TT_FILE_IN_SESSION if, for example, the ToolTalk database server is not installed on the file server that owns *pathname*.

The *ttdt_file_join*( ) function associates *the_scope* and a copy of *pathname* with the **Tt_pattern**s returned, so that *ttdt_file_quit*( ) can access them. Thus, the caller is free to modify or free *pathname* after *ttdt_file_join*( ) returns.

The *clientdata* argument points to arbitrary data that will be passed into the callback unmodified.

The **Ttdt_file_cb** argument is a callback defined as:

```
Tt_message (*Ttdt_file_cb)(Tt_message msg,
                           Tttk_op op,
                           char *pathname,
                           void *clientdata,
                           int same_euid_egid,
                           int same_procid);
```

The *message* argument is the message. The *op* argument is the operation. The *pathname* argument is the pathname of the file the message is about. The *clientdata* argument is the client data passed into *ttdt_file_join*( ). The *same_euid_egid* argument is True if the sender can be trusted; otherwise it is False. The *same_procid* argument is True if the sender is the same procid as the receiver; otherwise it is False. A **Ttdt_file_cb** must return the message if it does not consume the message. (Consuming means replying, rejecting or failing a request, and then destroying the message.) Otherwise, it must consume the message and return either zero or a *tt_error_pointer*( ) cast to **Tt_message**.

RETURN VALUE

Upon successful completion, the *ttdt_file_join*( ) function returns a null-terminated array of **Tt_pattern**, which can be passed to *ttdt_file_event*( ) to register for requests that the application should handle once it begins to modify the file; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PATH
: The specified pathname included an unsearchable directory.

APPLICATION USAGE

The null-terminated array of **Tt_pattern** returned by *ttdt_file_join*( ) should be destroyed by passing the array to *ttdt_file_quit*( ).

The *pathname* argument to **Ttdt_file_cb** is a copy that can be freed using *tt_free*( ).

EXAMPLES

This is the typical algorithm of a **Ttdt_file_cb**:

```
Tt_message myFileCB(Tt_message      msg,
    Tttk_op         op,
    char           *pathname,
    int             trust,
    int             isMe)
{
    tt_free(pathname);
    Tt_status status = TT_OK;
    switch(op) {
        case TTDT_MODIFIED:
        if ((_modifiedByMe)&&(! isMe)) {
            /* Hmm, the other editor either does not know or
             * does not care that we are already modifying the
             * file, so the last saver will win.
             */
        } else {
            /* Interrogate user if she ever modifies the buffer */
            _modifiedByOther = 1;
            XtAddCallback(myTextWidget, XmNmodifyVerifyCallback,
                      myTextModifyCB, 0);
        }
        break;
        case TTDT_GET_MODIFIED:
        tt_message_arg_ival_set(msg, 1, _modifiedByMe);
        tt_message_reply(msg);
        break;
        case TTDT_SAVE:
        status = mySave(trust);
        if (status == TT_OK) {
            tt_message_reply(msg);
        } else {
            tttk_message_fail(msg, status, 0, 0);
        }
```

```
            break;
            case TTDT_REVERT:
            status = myRevert(trust);
            if (status == TT_OK) {
                tt_message_reply(msg);
            } else {
                tttk_message_fail(msg, status, 0, 0);
            }
            break;
            case TTDT_REVERTED:
            if (! isMe) {
                _modifiedByOther = 0;
            }
            break;
            case TTDT_SAVED:
            if (! isMe) {
                _modifiedByOther = 0;
                int choice = myUserChoice(myContext, myBaseFrame,
                             "Another tool has saved "
                             "this file.", 2, "Ignore",
                             "Revert");
                switch(choice) {
                    case 1:
                    myRevert(1);
                    break;
                }
            }
            break;
            case TTDT_MOVED:
            case TTDT_DELETED:
            /* Do something appropriate */
            break;
        }
        tttk_message_destroy(msg);
        return 0;
    }
```

SEE ALSO

<Tt/tttk.h>, *ttdt_file_quit*( ), *ttdt_file_event*( ), *ttdt_Get_Modified*( ), *ttdt_Save*( ), *ttdt_Revert*( ), *tt_file_join*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_file_notice — create and send a standard ToolTalk notice about a file

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_message ttdt_file_notice(Tt_message context,
                            Tttk_op op,
                            Tt_scope scope,
                            const char *pathname,
                            int send_and_destroy);
```

DESCRIPTION

The *ttdt_file_notice*( ) function is used to create (and optionally send) any of the standard file notices: *Created*, *Deleted*, *Moved*, *Reverted*, *Saved*, and *Modified*.

The *ttdt_file_notice*( ) function creates a notice with the specified *op* and *scope*, and sets its file attribute to *pathname*. The function adds an unset argument of **Tt_mode** TT_IN and vtype *File* to the notice, per the Desktop messaging conventions. If *send_and_destroy* is True, *ttdt_file_notice*( ) sends the message and then destroys it; otherwise, it only creates the message.

The *context* argument describes the environment to use. If *context* is not zero, messages created by *ttdt_file_notice*( ) inherit from *context* all contexts whose slotname begins with the characters ENV_.

RETURN VALUE

If *send_and_destroy* is False, the *ttdt_file_notice*( ) function returns the created **Tt_message**. If *send_and_destroy* is True, it returns zero; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_DESKTOP_EINVAL
: The *op* argument was TTDT_MOVED and *send_and_destroy* was True.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_POINTER
: The *pathname* argument was NULL or was a ToolTalk error pointer.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

APPLICATION USAGE

The *ttdt_file_event*( ) function is a higher-level interface than *ttdt_file_notice*( ), and is the preferred way to send all but the *Moved* notice.

SEE ALSO

<Tt/tttk.h>, *ttdt_file_event*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_file_quit — unregister interest in ToolTalk events about a file

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttdt_file_quit(Tt_pattern *patterns,
                         int quit);
```

DESCRIPTION

The *ttdt_file_quit*( ) function is used to unregister interest in the *pathname* that was passed to *ttdt_file_join*( ) when *patterns* was created. The *ttdt_file_quit*( ) function destroys *patterns* and sets the default file to NULL.

If *quit* is True, *ttdt_file_quit*( ) calls *tt_file_quit*( ) with a *pathname* argument; otherwise, it returns without quitting.

RETURN VALUE

Upon successful completion, the *ttdt_file_quit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The *patterns* argument was NULL or otherwise invalid.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tttk.h>, *ttdt_file_join*( ), *tt_default_file*( ), *tt_file_quit*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_file_request — create and send a standard ToolTalk request about a file

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_message ttdt_file_request(Tt_message context,
                             Tttk_op op,
                             Tt_scope scope,
                             const char *pathname,
                             Ttdt_file_cb cb,
                             void *client_data,
                             int send_and_destroy);
```

DESCRIPTION

The *ttdt_file_request*( ) function is used to create (and optionally send) any of the standard Desktop file requests defined in Section 6.6 on page 364, such as *Get_Modified*, *Save*, and *Revert*.

The *ttdt_file_request*( ) function creates a request with the specified *op* and *scope*, and sets its file attribute to *pathname*. The function adds an unset argument of **Tt_mode** TT_IN and vtype *File* to the request, per the Desktop messaging conventions. If *op* is TTDT_GET_MODIFIED, *ttdt_file_request*( ) also adds an unset TT_OUT argument of vtype *Boolean* to the request. The *ttdt_file_request*( ) function installs *cb* as a message callback for the created request, and ensures that *client_data* will be passed into the callback. (The **Ttdt_file_cb** callback is described under *ttdt_file_join*( )). If *send* is True, *ttdt_file_request*( ) sends the request before returning the handle to it; otherwise, it only creates the request.

The *context* argument describes the environment to use. If *context* is not zero, messages created by *ttdt_file_request*( ) inherit from *context* all contexts whose slotname begins with the characters ENV_.

RETURN VALUE

Upon successful completion, the *ttdt_file_request*( ) function returns the created **Tt_message**; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_ERR_DBAVAIL
: The ToolTalk service could not access the ToolTalk database needed for this operation.

TT_ERR_DBEXIST
: The ToolTalk service could not access the specified ToolTalk database in the expected place.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_POINTER
: The *pathname* argument was NULL or was a ToolTalk error pointer.

TT_ERR_PROCID
The specified process identifier is out of date or invalid.

APPLICATION USAGE

The *ttdt_file_request*( ) function is a lower-level interface than *ttdt_Get_Modified*( ), *ttdt_Save*( ), and *ttdt_Revert*( ), since the latter functions create and send the request and then block on its reply.

SEE ALSO

<Tt/tttk.h>, *ttdt_Get_Modified*( ), *ttdt_Save*( ), *ttdt_Revert*( ), *ttdt_file_join*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_message_accept — accept a contract to handle a ToolTalk request

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_pattern *ttdt_message_accept(Tt_message contract,
                                Ttdt_contract_cb cb,
                                Widget shell,
                                void *clientdata,
                                int accept,
                                int sendStatus);
```

DESCRIPTION

The *ttdt_message_accept*( ) function registers in the default session for TT_HANDLER-addressed requests:

(1) *Get_Geometry*, *Set_Geometry*, *Get_Iconified*, *Set_Iconified*, *Get_Mapped*, *Set_Mapped*, *Raise*, *Lower*, *Get_XInfo*

(2) *Pause*, *Resume*

(3) *Quit*, *Get_Status*

If the *shell* argument is not NULL, the ToolTalk service handles messages in (1) transparently; otherwise, it treats them like messages in (3).

If *shell* is non-NULL and *cb* is NULL, then the ToolTalk service handles messages in (2) transparently by passing *shell* and the appropriate boolean value to *XtSetSensitive*( ). If *cb* is NULL, then the ToolTalk service treats messages in (2) like (3).

If *cb* is not NULL, *ttdt_message_accept*( ) passes messages in (3) to the *cb* callback; otherwise it fails with TT_DESKTOP_ENOTSUP.

If *accept* is True, *ttdt_message_accept*( ) calls *tt_message_accept*( ) with a *contract* argument. If *contract* has a returned value from *tt_message_status*( ) of TT_WRN_START_MESSAGE, it is the message that caused the tool to be started. The tool should join any scopes it wants to serve before accepting *contract*, so that it will receive any other messages already dispatched to its ptype. Otherwise, those messages will cause other instances of the ptype to be started. If that is in fact desired (for example, because the tool can only service one message at a time), then the tool should undeclare its ptype while it is busy.

If *sendStatus* is True, *ttdt_message_accept*( ) sends a *Status* notice to the requester, using the arguments (if any) passed to *ttdt_open*( ).

RETURN VALUE

Upon successful completion, the *ttdt_message_accept*( ) function returns a null-terminated array of **Tt_pattern**, and associates this array with *contract*; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_UNIMP

The *ttsession* for the default session is of a version that does not support *tt_message_accept*( ). If *contract* is a TT_WRN_START_MESSAGE, messages to the tool's ptype will remain blocked until *contract* is rejected, replied to, or failed.

APPLICATION USAGE

The *ttdt_message_accept*( ) function is what a tool calls when it wants to accept responsibility for handling (that is, failing or rejecting) a request.

If *contract* is destroyed by *tttk_message_destroy*( ), then the patterns will also be destroyed. Otherwise, the caller is responsible for iterating over the array and destroying each pattern.

EXAMPLES

See *ttdt_session_join*( ) for an example of a **Ttdt_contract_cb** callback that can be used with *ttdt_message_accept*( ).

SEE ALSO

<Tt/tttk.h>, *ttdt_open*( ), *ttmedia_ptype_declare*( ), *tt_ptype_declare*( ), *ttdt_session_join*( ), *ttdt_file_join*( ), *tt_ptype_undeclare*( ), *tt_ptype_undeclare*( ); *XtSetSensitive*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_open — create a ToolTalk communication endpoint

SYNOPSIS

```
#include <Tt/tttk.h>

char *ttdt_open(int *ttfd,
                const char *toolname,
                const char *vendor,
                const char *version,
                int sendStarted);
```

DESCRIPTION

The *ttdt_open*( ) function calls *tt_open*( ) and *tt_fd*( ). It associates *toolname*, *vendor* and *version* with the created procid, and initialises the new procid's default contexts from the process environment. If *sendStarted* is True, *ttdt_open*( ) sends a *Started* notice.

RETURN VALUE

Upon successful completion, the *ttdt_open*( ) function returns the created procid in a string that can be freed with *tt_free*( ); otherwise, the *ttdt_open*( ) function may return any of the errors returned by *tt_open*( ) and *tt_fd*( ).

No errors are propagated if sending the *Started* notice fails.

SEE ALSO

<Tt/tttk.h>, *ttdt_close*( ), *tt_open*( ), *tt_fd*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_sender_imprint_on — act like a child of the specified tool

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttdt_sender_imprint_on(const char *handler,
                                 Tt_message contract,
                                 char **display,
                                 int *width,
                                 int *height,
                                 int *xoffset,
                                 int *yoffset,
                                 XtAppContext app2run,
                                 int ms_timeout);
```

DESCRIPTION

The *ttdt_sender_imprint_on*( ) function is used to make the calling tool act equivalently to a child of another specified tool. The calling tool adopts the other tool's X11 display, locale, and current working directory. It also learns the other tool's X11 geometry, so that it may position itself appropriately.

If the *handler* argument is non-NULL, the requests are addressed to that procid using TT_HANDLER. If *handler* is NULL and the *contract* argument is non-NULL, the requests are addressed to the *tt_message_sender*( ) of the contract, using TT_HANDLER.

The *contract* argument is passed to *tttk_message_create*( ) as the *context* argument.

If the *display* argument is not NULL, *ttdt_sender_imprint_on*( ) returns the other tool's display in **display*. If *display* is NULL, *ttdt_sender_imprint_on*( ) sets the *DISPLAY* environment variable to the other tool's display.

If each of the *width*, *height*, *xoffset*, and *yoffset* arguments are NULL, then *ttdt_sender_imprint_on*( ) does not send the other tool a *Get_Geometry* request.

The *ttdt_sender_imprint_on*( ) function passes the *app2run* and *ms_timeout* arguments to *tttk_block_while*( ), blocking on the replies to the requests it sends.

If the *display* argument is not NULL, *ttdt_sender_imprint_on*( ) sets **display* to a string that can be freed with *tt_free*( ).

If for some reason no width or height is returned by the other tool, *ttdt_sender_imprint_on*( ) sets **width* or **height* to –1. If no positional information is returned, *ttdt_sender_imprint_on*( ) sets **xoffset* and **yoffset* to {INT_MAX}.

RETURN VALUE

Upon successful completion, the *ttdt_sender_imprint_on*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_DESKTOP_ETIMEDOUT
: One or more of the sent requests did not complete within *ms_timeout* milliseconds.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

APPLICATION USAGE

If both the *handler* and *contract* are zero, the requests are addressed to no tool in particular, using TT_PROCEDURE; this is not recommended.

SEE ALSO

<Tt/tttk.h>, *tt_free*( ), *tt_message_sender*( ), *tttk_block_while*( ), *tttk_message_create*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_session_join — join a ToolTalk session

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_pattern *ttdt_session_join(const char *sessid,
                              Ttdt_contract_cb cb,
                              Widget shell,
                              void *clientdata,
                              int join);
```

DESCRIPTION

The *ttdt_session_join*( ) function joins the session *sessid*, registering patterns and default callbacks for many standard Desktop message interfaces. If *sessid* is NULL, the default session is joined.

The *ttdt_session_join*( ) function registers for the following TT_HANDLER-addressed requests:

(1) *Get_Environment*, *Set_Environment*, *Get_Locale*, *Set_Locale*, *Get_Situation*, *Set_Situation*, *Signal*, *Get_Sysinfo*

(2) *Get_Geometry*, *Set_Geometry*, *Get_Iconified*, *Set_Iconified*, *Get_Mapped*, *Set_Mapped*, *Raise*, *Lower*, *Get_XInfo*

(3) *Pause*, *Resume*, *Quit*

(4) *Get_Status*, *Do_Command*

If *join* is True, *ttdt_session_join*( ) actually joins the indicated session.

The ToolTalk service handles messages in (1) transparently.

If *shell* is non-NULL, then it is expected to be a realised *mappedWhenManaged applicationShellWidget*, and the ToolTalk service handles messages in (2) transparently. (If *shell* is merely a realised widget, then the ToolTalk service handles only the *Get_XInfo* request, and *ttdt_session_join*( ) fails the rest of (2) with TT_DESKTOP_ENOTSUP.) If *shell* is NULL, then the ToolTalk service treats messages in (2) equivalently to those in (4).

If *shell* is non-NULL and *cb* is NULL, then the ToolTalk service handles messages in (3) transparently as follows; otherwise, it treats them as equivalent to those in (4). The *Quit* request results in a WM_DELETE_WINDOW event on *shell* if the *silent* and *force* arguments of the *Quit* request are both False. In other words, if *shell* is supplied without a *cb*, then a *Quit* request may imply that the user quit the application's top-level window using the window manager. *Pause* and *Resume* requests result in the ToolTalk service passing *shell* and the appropriate boolean value to *XtSetSensitive*( ).

If *cb* is not NULL, the ToolTalk service passes messages in (4) to *cb*; otherwise, *ttdt_session_join*( ) fails with TT_DESKTOP_ENOTSUP.

The **Ttdt_contract_cb** argument is a callback defined as:

```
Tt_message (*Ttdt_contract_cb)(Tt_message msg,
                               void *clientdata,
                               Tt_message contract);
```

The *msg* argument is a message in **Tt_state** TT_SENT. If *msg* is a TT_REQUEST, the client program becomes responsible for either failing, rejecting or replying to *msg*. After doing so, the client program may dispose of *msg* with *tttk_message_destroy*( ). The *clientdata* argument is the *clientdata* passed to *ttdt_session_join*( ) or *ttdt_message_accept*( ). The *contract* argument is the *contract* passed to *ttdt_message_accept*( ). For callbacks installed by *ttdt_session_join*( ), *contract* is always zero.

RETURN VALUE

Upon successful completion, the *ttdt_session_join*( ) function returns a null-terminated array of **Tt_pattern**; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

APPLICATION USAGE

The null-terminated array of **Tt_pattern** returned by *ttdt_session_join*( ) should be destroyed by passing the array to *ttdt_file_quit*( ).

The ToolTalk service will reply to the *Quit* request before generating the WM_DELETE_WINDOW event. If the application catches and cancels this event, then the sender of the *Quit* request will be misled into thinking the application actually quit. Applications that can cancel WM_DELETE_WINDOW should install a real **Ttdt_contract_cb**.

The ToolTalk service handles the *Pause* and *Resume* requests by setting the sensitivity of *widget*. If *widget* is the parent of any top-level pop-up shells, *XtSetSensitive*( ) will not affect them. Applications that can have such pop-ups should install a real **Ttdt_contract_cb**.

A **Ttdt_contract_cb** should return zero if it processes *msg* successfully, or a *tt_error_pointer*( ) cast to **Tt_message** if processing results in an error. It should return the *msg* if it does not consume it. If *msg* is returned, then the ToolTalk service passes TT_CALLBACK_CONTINUE down the call stack, so that *msg* will be offered to other callbacks or (more likely) be returned from *tt_message_receive*( ). Applications will rarely want *msg* to get processed by other callbacks or in the main event loop.

EXAMPLES

This is the typical algorithm of a **Ttdt_contract_cb** for an application that handles *Pause*, *Resume* or *Quit* requests for itself, but lets the ToolTalk service handle the X11-related requests listed in (2). Since this example callback deals with the case when *contract* is not zero, it can also be used as the **Ttdt_contract_cb** passed to *ttdt_message_accept*( ).

```
Tt_message myContractCB(Tt_message      msg,
       void           *clientdata,
       Tt_message      contract)
{
```

```
char *opString = tt_message_op(msg);
Tttk_op op = tttk_string_op(opString);
tt_free(opString);
int silent = 0;
int force  = 0;
Boolean cancel = False;
Boolean sensitive = True;
char *status, command;
switch(op) {
    case TTDT_QUIT:
       tt_message_arg_ival(msg, 0, &silent);
       tt_message_arg_ival(msg, 1, &force);
       if (contract == 0) {
              /* Quit entire application */
              cancel = ! myQuitWholeApp(silent, force);
       } else {
              /* Quit just the specified request being
                 worked on */
              cancel = ! myCancelThisRequest(contract,
                     silent, force);
       }
       if (cancel) {
              /* User canceled Quit; fail the Quit request */
              tttk_message_fail(msg, TT_DESKTOP_ECANCELED, 0, 1);
       } else {
              tt_message_reply(msg);
              tttk_message_destroy(msg);
       }
       return 0;
    case TTDT_PAUSE:
       sensitive = False;
    case TTDT_RESUME:
       if (contract == 0) {
              int already = 1;
              if (XtIsSensitive(myTopShell) != sensitive) {
                     already = 0;
                     XtSetSensitive(myTopShell, sensitive);
              }
              if (already) {
                     tt_message_status_set(msg,
                            TT_DESKTOP_EALREADY);
              }
       } else {
              if (XtIsSensitive(thisShell) == sensitive) {
                     tt_message_status_set(msg,
                            TT_DESKTOP_EALREADY);
              } else {
                     XtSetSensitive(thisShell, sensitive);
              }
       }
       tt_message_reply(msg);
       tttk_message_destroy(msg);
```

```
                return 0;
            case TTDT_GET_STATUS:
                if (contract == 0) {
                        status = "Message about status of entire app";
                } else {
                        status = "Message about status of this request";
                }
                tt_message_arg_val_set(msg, 0, status);
                tt_message_reply(msg);
                tttk_message_destroy(msg);
                return 0;
            case TTDT_DO_COMMAND:
                if (! haveExtensionLanguage) {
                        tttk_message_fail(msg, TT_DESKTOP_ENOTSUP, 0, 1);
                        return 0;
                }
                command = tt_message_arg_val(msg, 0);
                result = myEval(command);
                tt_free(command);
                tt_message_status_set(msg, result);
                if (tt_is_err(result)) {
                        tttk_message_fail(msg, result, 0, 1);
                } else {
                        tt_message_reply(msg);
                        tttk_message_destroy(msg);
                }
                return 0;
        }
        /* Unrecognized message; do not consume it */
        return msg;
}
```

SEE ALSO

<Tt/tttk.h>, *ttdt_session_quit*( ), *tt_session_join*( ), *XtSetSensitive*( ) in the X / Open CAE Specification, **Window Management**: **X Toolkit Intrinsics**,

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_session_quit — quit a ToolTalk session

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttdt_session_quit(const char *sessid,
                            Tt_pattern *sess_pats,
                            int quit);
```

DESCRIPTION

The *ttdt_session_quit*( ) function destroys the patterns in *sess_pats*. If *quit* is True, it quits the session *sessid*, or the default session if *sessid* is NULL.

RETURN VALUE

Upon successful completion, the *ttdt_session_quit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

TT_ERR_SESSION
: The specified ToolTalk session is out of date or invalid.

SEE ALSO

<Tt/tttk.h>, *ttdt_session_join*( ), *tt_session_quit*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttdt_subcontract_manage — manage an outstanding request

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_pattern *ttdt_subcontract_manage(Tt_message subcontract,
                                    Ttdt_contract_cb cb,
                                    Widget shell,
                                    void *clientdata);
```

DESCRIPTION

The *ttdt_subcontract_manage*( ) function allows a requester to manage the standard Desktop interactions with the tool that is handling the request. The *ttdt_subcontract_manage*( ) function registers in the default session for TT_HANDLER-addressed requests *Get_Geometry* and *Get_XInfo*, and *Status* notices.

If *shell* is not NULL, the ToolTalk service handles the *Get_Geometry* and *Get_XInfo* notices transparently; otherwise, it passes them to *cb*. The *Status* notice is always passed to the callback.

See *ttdt_session_join*( ) for a description of a **Ttdt_contract_cb** callback.

If *subcontract* is destroyed by *tttk_message_destroy*( ), then the patterns will also be destroyed; otherwise, the caller is responsible for iterating over the array and destroying each pattern.

RETURN VALUE

Upon successful completion, the *ttdt_subcontract_manage*( ) function returns a null-terminated array of **Tt_pattern**, and associates this array with *subcontract*; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_DESKTOP_EINVAL
: Both the *shell* and *cb* arguments were NULL.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The *subcontract* argument was not a valid **Tt_message**.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tttk.h>, *ttdt_session_join*( ), *tttk_message_destroy*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttmedia_Deposit — send a Deposit request to checkpoint a document

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttmedia_Deposit(Tt_message load_contract,
                          const char *buffer_id,
                          const char *media_type,
                          const unsigned char *new_contents,
                          int new_len,
                          const char *pathname,
                          XtAppContext app2run,
                          int ms_timeout);
```

DESCRIPTION

The *ttmedia_Deposit*( ) function is used to perform a checkpoint save on a document that was the subject of a Media Exchange *load_contract* request such as *Edit*, *Compose*, or *Open*. (See Section 6.6.2 on page 396.) To carry out a checkpoint save, the editor must send the new document contents back to the sender of *load_contract*.

The *ttmedia_Deposit*( ) function creates and sends a *Deposit* request and returns the success or failure of that request. The *load_contract* argument is the request that caused this editor to load the document. The *buffer_id* argument is the identifier of the buffer this editor created if the document was loaded via an *Open* request. If *buffer_id* is NULL, the the ToolTalk service gives the *Deposit* request a **Tt_address** of TT_HANDLER and sends it directly to the *tt_message_sender*( ) of *load_contract*; otherwise, the the ToolTalk service will address it as a TT_PROCEDURE and insert *buffer_id* into the request to match the pattern registered by the sender of the *load_contract*.

The *ttmedia_Deposit*( ) function uses the *media_type* argument as the vtype of the contents argument of the sent request, and *new_contents* and *new_len* as its value. The latter two must be zero if *pathname* is used to name a temporary file into which the editor will place the checkpointed document. The editor is free to remove the temporary file after the reply to the *Deposit* request is received; that is, after *ttmedia_Deposit*( ) has returned.

After the request is sent, *ttmedia_Deposit*( ) passes *app2run* and *ms_timeout* to *tttk_block_while*( ) to wait for the reply.

RETURN VALUE

Upon successful completion, the *ttmedia_Deposit*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_DESKTOP_ETIMEDOUT
: No reply was received within *ms_timeout* milliseconds.

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_POINTER
: The *pathname* argument was NULL or was a ToolTalk error pointer.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tttk.h>, *ttmedia_load*( ), *ttmedia_load_reply*( ), *ttmedia_ptype_declare*( ), *ttdt_Save*( ), *tttk_block_while*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttmedia_load — send a Display, Edit or Compose request

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_message ttmedia_load(Tt_message context,
                        Ttmedia_load_msg_cb cb,
                        void *clientdata,
                        Tttk_op op,
                        const char *media_type,
                        const unsigned char *contents,
                        int len,
                        const char *file,
                        const char *docname,
                        int send);
```

DESCRIPTION

The *ttmedia_load*( ) function is used to create and optionally send a Media Exchange request to display, edit or compose a document.

The *cb* argument will be passed *clientdata* when the reply is received, or when intermediate versions of the document are checkpointed through *Deposit* requests. The *op* argument must be one of TTME_DISPLAY, TTME_EDIT or TTME_COMPOSE. The *media_type* argument names the data format of the document, and is usually the primary determinant of which application will be chosen to handle the request. The *contents* and *len* arguments specify the document; if they are NULL and zero, respectively, and *file* is not NULL, then the document is assumed to be contained in *file*. If *docname* is not NULL, then *ttmedia_load*( ) uses it as the title of the document. If *send* is True, the message is sent before being returned.

The *context* argument describes the environment to use. If *context* is not zero, messages created by *ttmedia_load*( ) inherit from *context* all contexts whose slotname begins with the characters ENV_.

The **Ttmedia_load_msg_cb** argument is a callback defined as:

```
Tt_message (*Ttmedia_load_msg_cb)(Tt_message msg,
                                  void *clientdata),
                                  Tttk_op op,
                                  unsigned char *contents,
                                  int len,
                                  char *file);
```

The *msg* argument is the reply to the load request, or a *Deposit* request with a *messageID* argument naming the identifier (see *tt_message_id*( )) of the load request. In the latter case, the client program becomes responsible for either failing or replying to the request. In either case, *msg* should be destroyed after being processed.

The *op* argument is the op of *msg*. It must be either TTME_DEPOSIT or the *op* passed to *ttmedia_load*( ).

The *contents*, *len* and *file* arguments represent the contents of the arriving document. If *len* is zero, then the document is contained in *file*. If *contents* or *file* are non-NULL, they can be freed using *tt_free*( ).

The *clientdata* argument is the *clientdata* passed to *ttmedia_load*( ).

RETURN VALUE

Upon successful completion, the *ttmedia_load*( ) function returns the request it was asked to build; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_OVERFLOW
: The ToolTalk service has more active messages than it can handle. (The maximum number of active messages is implementation specific, but is at least 2000.)

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

APPLICATION USAGE

After the request created by *ttmedia_load*( ) is sent, the application will probably want to use *ttdt_subcontract_manage*( ) immediately afterwards to manage the standard interactions with the handler of the request.

A **Ttmedia_load_msg_cb** callback should return NULL if it processes *msg* successfully, or a *tt_error_pointer*( ) cast to **Tt_message** if processing results in an error. It should return the *msg* if it does not consume it, in which case the ToolTalk service will pass TT_CALLBACK_CONTINUE down the call stack, so that *msg* will be offered to other callbacks or (more likely) be returned from *tt_message_receive*( ). Applications will rarely want *msg* to get processed by other callbacks or in the main event loop.

EXAMPLES

This is the typical algorithm of a **Ttmedia_load_msg_cb**:

```
Tt_message
myLoadMsgCB(Tt_message  msg,
        void           *clientData,
        Tttk_op         op,
        unsigned char  *contents,
        int             len,
        char           *file)
{
        if (len > 0) {
                /* Replace data with len bytes in contents */
        } else if (file != 0) {
                /* Replace data with data read from file */
        }
        if (op == TTME_DEPOSIT) {
                tt_message_reply(msg);
        }
        tttk_message_destroy(msg);
        return 0;
}
```

SEE ALSO

<Tt/tttk.h>, *ttmedia_load_reply* ( ), *ttmedia_ptype_declare* ( ), *ttmedia_Deposit* ( ), *tt_free* ( ), *tt_message_receive* ( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttmedia_load_reply — reply to a Display, Edit or Compose request

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_message ttmedia_load_reply(Tt_message contract,
                              const unsigned char *new_contents,
                              int new_len,
                              int reply_and_destroy);
```

DESCRIPTION

The *ttmedia_load_reply* ( ) function is used to reply to a Media Exchange request to display, edit or compose a document. The editor working on the request usually calls *ttmedia_load_reply* ( ) when the user has indicated in some way that he or she is finished viewing or modifying the document.

If *new_contents* and *new_len* are non-NULL and non-zero, respectively, *ttmedia_load_reply* ( ) uses their values to set the new contents of the document back in the appropriate output argument of *contract*. If *reply_and_destroy* is True, *ttmedia_load_reply* ( ) replies to *contract* and then destroys it.

RETURN VALUE

Upon successful completion, the *ttmedia_load_reply* ( ) function returns the created **Tt_message**; otherwise, it returns an error pointer. The application can use *tt_ptr_error* ( ) to extract one of the following **Tt_status** values from the returned handle:

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NOTHANDLER
: This application is not the handler for this message.

TT_ERR_NUM
: The integer value passed was invalid (out of range).

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

APPLICATION USAGE

If *contract* is a *Display* request, then *new_contents* and *new_len* should be zero.

SEE ALSO

<Tt/tttk.h>, *ttmedia_load* ( ), *ttmedia_ptype_declare* ( ), *ttmedia_Deposit* ( ).

CHANGE HISTORY

First released in Issue 1.

NAME

ttmedia_ptype_declare — declare the ptype of a Media Exchange media editor

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status ttmedia_ptype_declare(const char *ptype,
                                int base_opnum,
                                Ttmedia_load_pat_cb cb,
                                void *clientdata,
                                int declare);
```

DESCRIPTION

The *ttmedia_ptype_declare*( ) function is used to initialise an editor that implements the Media Exchange message interface for a particular media type. The *ttmedia_ptype_declare*( ) function notifies the ToolTalk service that the *cb* callback is to be called when the editor is asked to edit a document of the kind supported by ptype. The *ttmedia_ptype_declare*( ) function installs an implementation-specific *opnum* callback on a series of signatures that ptype is assumed to contain. These signatures are listed below, with their corresponding opnum offsets. Opnums in ptype for these signatures start at *base_opnum*, which must be zero or a multiple of 1000. The implementation-specific opnum callback will pass *clientdata* to *cb* when a request is received that matches one of these signatures.

If *declare* is True, *ttmedia_ptype_declare*( ) calls *tt_ptype_declare*( ) with the ptype argument. If ptype implements Media Exchange for several different media types, then *ttmedia_ptype_declare*( ) can be called multiple times, with a different *base_opnum* each time, and with *declare* being True only once.

The **Ttmedia_load_pat_cb** argument is a callback defined as:

```
Tt_message (*Ttmedia_load_pat_cb)(Tt_message msg,
                                  void *clientdata,
                                  Tttk_op op,
                                  Tt_status diagnosis,
                                  unsigned char *contents,
                                  int len,
                                  char *file,
                                  char *docname);
```

The *msg* argument is a TT_REQUEST in **Tt_state** TT_SENT. The client program becomes responsible for either failing, rejecting or replying to it. This can either be done inside the callback, or the message can be saved and dismissed later (that is, after the callback returns). Usually, the callback will either immediately reject / fail the request, or it will start processing the request, perhaps by associating it with a new window. When the request is finally dismissed, it should be destroyed, for example, using *tt_message_destroy*( ).

If the callback knows it will handle the request (either fail or reply to it, but not reject it), then it should call *ttdt_message_accept*( ). But if the return value of *tt_message_status*( ) of *msg* is TT_WRN_START_MESSAGE, then the callback should probably do *ttdt_session_join*( ), and perhaps a *ttdt_file_join*( ), before accepting the message. The *op* argument is the op of the incoming request, one of TTME_COMPOSE, TTME_EDIT or TTME_DISPLAY. The *diagnosis* argument is the recommended error code; if the ToolTalk service detects a problem with the request (for example, TT_DESKTOP_ENODATA), then it passes in the error code that it recommends the request should be failed with. If *diagnosis* was not TT_OK and the **Ttmedia_load_pat_cb** returns *msg*, then the ToolTalk service will fail and destroy *msg*.

The ToolTalk service does not simply fail the request transparently, because the request may be the reason that the client process was started by ToolTalk in the first place. So if *diagnosis* is not TT_OK and the *tt_message_status*( ) of *msg* is TT_WRN_START_MESSAGE, then many applications will decide that they have no reason to continue running. If such an application chooses to exit in the callback, then it should first dismiss the request. Otherwise, it can set some global flag, return *msg* (thus allowing the ToolTalk service to dismiss the message), and then have *main*( ) check the flag and exit before even entering the event loop. (Exiting without dismissing the request would fail it with status TT_ERR_PROCID, instead of with *diagnostic*.)

The *contents*, *len*, and *file* arguments represent the contents of the arriving document. If *len* is zero, then the document is contained in *file*. If *contents* or *file* are non-NULL, they can be freed using *tt_free*( ).

The *docname* argument is the name of the document, if any. The *clientdata* argument is the *clientdata* passed to *ttmedia_ptype_declare*( ).

A **Ttmedia_load_pat_cb** should return zero if it processes *msg* successfully, or a *tt_error_pointer*( ) cast to **Tt_message** if processing results in an error. It should return the *msg* if it does not consume it. If *diagnosis* is not TT_OK and *msg* is returned, then the ToolTalk service will consume (namely, fail and destroy) it. If *diagnosis* is TT_OK and *msg* is returned, then the ToolTalk service will pass TT_CALLBACK_CONTINUE down the call stack, so that *msg* will be offered to other callbacks or (more likely) be returned from *tt_message_receive*( ). Applications will rarely want *msg* to get processed by other callbacks or in the main event loop.

RETURN VALUE

Upon successful completion, the *ttmedia_ptype_declare*( ) function returns the status of the operation. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

TT_ERR_PTYPE
: The specified process type is not the name of an installed process type.

EXAMPLES

This is the typical algorithm of a **Ttmedia_load_pat_cb**:

```
Tt_message
myAcmeSheetLoadCB(
        Tt_message       msg,
        void            *client_data,
        Tttk_op          op,
        Tt_status        diagnosis,
        unsigned char   *contents,
        int              len,
        char            *file,
        char            *docname
)
{
```

```
        Tt_status status = TT_OK;
        if (diagnosis != TT_OK) {
                /* toolkit detected an error */
                if (tt_message_status(msg) == TT_WRN_START_MESSAGE) {
                        /*
                         * Error is in start message!  We now have no
                         * reason to live, so tell main() to exit().
                         */
                        myAbortCode = 2;
                }
                /* let toolkit handle the error */
                return msg;
        }
        if ((op == TTME_COMPOSE)&&(file == 0)) {
                /* open empty new buffer */
        } else if (len > 0) {
                /* load contents into new buffer */
        } else if (file != 0) {
                if (ttdt_Get_Modified(msg, file, TT_BOTH, myCntxt, 5000)) {
                        switch(myUserChoice("Save, Revert, Ignore?")) {
                            case 0:
                                ttdt_Save(msg, file, TT_BOTH, myCntxt, 5000);
                                break;
                            case 1:
                                ttdt_Revert(msg, file, TT_BOTH, myCntxt, 5000);
                                break;
                        }
                }
                /* load file into new buffer */
        } else {
                tttk_message_fail(msg, TT_DESKTOP_ENODATA, 0, 1);
                tt_free(contents); tt_free(file); tt_free(docname);
                return 0;
        }
        int w, h, x, y = INT_MAX;
        ttdt_sender_imprint_on(0, msg, 0, &w, &h, &x, &y, myCntxt, 5000);
        positionMyWindowRelativeTo(w, h, x, y);
        if (maxBuffersAreNowOpen) {
                /* Un-volunteer to handle future requests until less busy */
                tt_ptype_undeclare("Acme_Calc");
        }
        if (tt_message_status(msg) == TT_WRN_START_MESSAGE) {
                /*
                 * Join session before accepting start message,
                 * to prevent unnecessary starts of our ptype
                 */
                ttdt_session_join(0, myContractCB, myShell, 0, 1);
        }
        ttdt_message_accept(msg, myContractCB, myShell, 0, 1, 1);
        tt_free(contents); tt_free(file); tt_free(docname);
        return 0;
}
```

This is the signature layout to which ptype should conform:

```
ptype Acme_Calc {
    start "acalc";
    handle:
        /*
         * Display Acme_Sheet
         * Include in tool's ptype if tool can display a document.
         */
        session Display( in    Acme_Sheet  contents     ) => start opnum = 1;
        session Display( in    Acme_Sheet  contents,
                         in    messageID   counterfoil  ) => start opnum = 2;
        session Display( in    Acme_Sheet  contents,
                         in    title       docName      ) => start opnum = 3;
        session Display( in    Acme_Sheet  contents,
                         in    messageID   counterfoil,
                         in    title       docName      ) => start opnum = 4;
        /*
         * Edit Acme_Sheet
         * Include in tool's ptype if tool can edit a document.
         */
        session Edit(    inout Acme_Sheet  contents     ) => start opnum = 101;
        session Edit(    inout Acme_Sheet  contents,
                         in    messageID   counterfoil  ) => start opnum = 102;
        session Edit(    inout Acme_Sheet  contents,
                         in    title       docName      ) => start opnum = 103;
        session Edit(    inout Acme_Sheet  contents,
                         in    messageID   counterfoil,
                         in    title       docName      ) => start opnum = 104;
        /*
         * Compose Acme_Sheet
         * Include in tool's ptype if tool can compose a document from scratch.
         */
        session Edit(    out   Acme_Sheet  contents     ) => start opnum = 201;
        session Edit(    out   Acme_Sheet  contents,
                         in    messageID   counterfoil  ) => start opnum = 202;
        session Edit(    out   Acme_Sheet  contents,
                         in    title       docName      ) => start opnum = 203;
        session Edit(    out   Acme_Sheet  contents,
                         in    messageID   counterfoil,
                         in    title       docName      ) => start opnum = 204;
        /*
         * Mail Acme_Sheet
         * Include in tool's ptype if tool can mail a document.
         */
        session Mail(    in    Acme_Sheet  contents     ) => start opnum = 301;
        session Mail(    inout Acme_Sheet  contents     ) => start opnum = 311;
        session Mail(    inout Acme_Sheet  contents,
                         in    title       docName      ) => start opnum = 313;
        session Mail(    out   Acme_Sheet  contents     ) => start opnum = 321;
        session Mail(    out   Acme_Sheet  contents,
                         in    messageID   counterfoil  ) => start opnum = 323;
};
```

SEE ALSO

<Tt/tttk.h>, *tt_ptype_declare*( ), *tt_ptype_undeclare*( ), *ttdt_message_accept*( ), *ttdt_session_join*( ), *ttdt_file_join*( ), *tt_free*( ), *tt_message_receive*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_Xt_input_handler — Process ToolTalk events for Xt clients

SYNOPSIS

```
#include <Tt/tttk.h>

void tttk_Xt_input_handler(XtPointer procid,
                           int *source,
                           XtInputId *id);
```

DESCRIPTION

If *procid* is not NULL, *tttk_Xt_input_handler*( ) passes it to *tt_default_procid_set*( ). The *tttk_Xt_input_handler*( ) function then calls *tt_message_receive*( ), which retrieves the next message available, if any, for the default procid. If *tt_message_receive*( ) returns TT_ERR_NOMP, then *tttk_Xt_input_handler*( ) closes the default procid with *ttdt_close*( ), and removes the input source **id* with *XtRemoveInput*( ) if *id* is not zero. If a message is available and *tt_message_receive*( ) returns it (indicating it was not consumed by any message or pattern callback), then the ToolTalk service passes the message to *tttk_message_abandon*( ).

RETURN VALUE

The *tttk_Xt_input_handler*( ) function returns no value.

APPLICATION USAGE

The application should use *tttk_Xt_input_handler*( ) as its Xt input handler unless some messages are expected not to be consumed by callbacks. (The only messages that absolutely cannot be intercepted and consumed by callbacks are those that match observe signatures in a ptype or otype.)

EXAMPLES

```
int myTtFd;
char *myProcID;
myProcID = ttdt_open(&myTtFd, "WhizzyCalc", "Acme", "1.0", 1);
/* ... */
/* Process the message that started us, if any */
tttk_Xt_input_handler(myProcID, 0, 0);
/* ... */
XtAppAddInput(myContext, myTtFd, (XtPointer)XtInputReadMask,
              tttk_Xt_input_handler, myProcID);
```

SEE ALSO

<Tt/tttk.h>, *ttdt_close*( ), *tttk_message_abandon*( ), *tt_default_procid_set*( ), *tt_message_receive*( ), *XtAppAddInput*( ), *XtRemoveInput*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_block_while — block while a counter is greater than zero

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status tttk_block_while(XtAppContext app2run,
                           const int *blocked,
                           int ms_timeout);
```

DESCRIPTION

The *tttk_block_while*( ) function is used to process asynchronous events, such as ToolTalk messages or window system events, while waiting for a condition or timeout.

If *app2run* is not zero, then an event loop is run for that application context, by repeatedly calling *XtAppProcessEvent*( ) with *ms_timeout* being effected using *XtAppAddTimeOut*( ). If *app2run* is zero, then the file descriptor (as returned by *tt_fd*( )) of the default procid is polled (using the *poll*( ) function) and *tttk_Xt_input_handler*( ) is called whenever the file descriptor is active.

If *blocked* is zero, then *tttk_block_while*( ) runs until *ms_timeout* occurs. If *blocked* is non-zero, then the loop is run until either *ms_timeout* occurs, or **blocked* is less than 1.

If *ms_timeout* is zero, *tttk_block_while*( ) checks once for events, processes the first one, and then returns. If *ms_timeout* is negative, no timeout is in effect.

RETURN VALUE

Upon successful completion, the *tttk_block_while*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_DESKTOP_ETIMEDOUT
: The timeout occurred within *ms_timeout* milliseconds, or *ms_timeout* was zero and no input was available.

TT_DESKTOP_EINTR
: The *app2run* argument was zero, and *poll*( ) was interrupted by a signal.

TT_DESKTOP_EAGAIN
: The *app2run* argument was zero, and *poll*( ) returned EAGAIN.

If *app2run* is not zero and *ms_timeout* is negative, then *tttk_block_while*( ) will only return when **blocked* is less than 1, with TT_OK being returned.

If *app2run* is not zero, *ms_timeout* is negative, and *blocked* is zero, then *tttk_block_while*( ) behaves equivalent to *XtAppMainLoop*( ), and will never return.

APPLICATION USAGE

If *app2run* is zero, then only messaging events for the default procid will be serviced. Events for other procids will be blocked, as will window system events, so that the graphical user interface of the application will not update itself even, for example, after expose events.

On the other hand, if the application passes its Xt context in as *app2run*, then window system events will continue to be handled, as will message activity for all procids for which an *XtAppAddInput*( ) has been done. Since the window system event loop is fully operational in this case, the application should take care to disable any user interface controls that the user should not operate while the application is waiting for *tttk_block_while*( ) to return.

SEE ALSO

<Tt/tttk.h>, *tttk_Xt_input_handler*( ); *poll*( ) in the X / Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**; *XtAppPending*( ), *XtAppAddTimeOut*( ), *XtAppNextEvent*( ), *XtDispatchEvent*( ) in the X / Open CAE Specification, **Window Management: X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_message_abandon — finalise a message properly

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status tttk_message_abandon(Tt_message msg);
```

DESCRIPTION

The *tttk_message_abandon*( ) function is used by an application when it does not understand a message and wants to get rid of it. The *tttk_message_abandon*( ) function fails or rejects *msg* if appropriate, and then destroys it. The *tttk_message_abandon*( ) will reject or fail the message only if *msg* is a TT_REQUEST in **Tt_state** TT_SENT. If it has a **Tt_address** of TT_HANDLER or a *tt_message_status*( ) of TT_WRN_START_MESSAGE, then it fails the message; otherwise, it rejects it. In either case, *tttk_message_abandon*( ) gives *msg* a message status (see *tt_message_status*( )) of TT_DESKTOP_ENOTSUP.

RETURN VALUE

Upon successful completion, the *tttk_message_abandon*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NOTHANDLER
: This application is not the handler for this message.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tttk.h>, *tt_message_status*( ), *tttk_message_fail*( ), *tttk_message_reject*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_message_create — create a message conforming to the XCDE conventions

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_message tttk_message_create(Tt_message context,
                               Tt_class the_class,
                               Tt_scope the_scope,
                               const char *handler,
                               const char *op,
                               Tt_message_callback callback);
```

DESCRIPTION

The *tttk_message_create*( ) function creates a message that propagates inherited contexts from one message to another. The *tttk_message_create*( ) function creates a message and copies onto it all the context slots from *context* whose slotname begins with the characters ENV_. It gives the created message a **Tt_class** of *the_class* and a **Tt_scope** of *the_scope*. If *handler* is not NULL, then *tttk_message_create*( ) addresses the message as a TT_HANDLER to that procid; otherwise, it gives the message a **Tt_address** of TT_PROCEDURE. It sets the message's op to *op* if *op* is not NULL. If *callback* is not NULL, *tttk_message_create*( ) adds it to the message as a message callback.

RETURN VALUE

Upon successful completion, the *tttk_message_create*( ) function returns the created **Tt_message**, which can be modified, sent, and destroyed like any other **Tt_message**; otherwise, it returns an error pointer. The application can use *tt_ptr_error*( ) to extract one of the following **Tt_status** values from the returned handle:

TT_ERR_NOMEM
: There is insufficient memory available to perform the function.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_PROCID
: The specified process identifier is out of date or invalid.

SEE ALSO

<Tt/tttk.h>, *tt_message_create*( ), *tttk_message_create*( ), *ttdt_file_notice*( ), *ttdt_file_request*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_message_destroy — destroy a message conforming to the XCDE conventions

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status tttk_message_destroy(Tt_message msg);
```

DESCRIPTION

The *tttk_message_destroy*( ) function can be used in place of *tt_message_destroy*( ). It destroys any patterns that may have been stored on *msg* by *ttdt_message_accept*( ) or *ttdt_subcontract_manage*( ). Then it passes *msg* to *tt_message_destroy*( ).

RETURN VALUE

Upon successful completion, the *tttk_message_destroy*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

TT_WRN_STOPPED
: The message is not actually destroyed. (A message is not destroyed if it is in a non-final state; for example, a request for which the reply has not been received.)

SEE ALSO

<Tt/tttk.h>, *tt_message_create*( ), *tt_message_destroy*( ), *ttdt_file_notice*( ), *ttdt_file_request*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_message_fail — fail a message

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status tttk_message_fail(Tt_message msg,
                            Tt_status status,
                            const char *status_string,
                            int destroy);
```

DESCRIPTION

The *tttk_message_fail*( ) function sets the status and status string of the TT_REQUEST *msg*, fails *msg*, and then destroys *msg* if *destroy* is True.

RETURN VALUE

Upon successful completion, the *tttk_message_fail*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NOTHANDLER
: This application is not the handler for this message.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tttk.h>, *tt_message_fail*( ), *tttk_message_abandon*( ), *tttk_message_reject*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_message_reject — reject a message

SYNOPSIS

```
#include <Tt/tttk.h>

Tt_status tttk_message_reject(Tt_message msg,
                              Tt_status status,
                              const char *status_string,
                              int destroy);
```

DESCRIPTION

The *tttk_message_reject*( ) function sets the status and status string of the TT_REQUEST *msg*, rejects the *msg*, and then destroys *msg* if *destroy* is True.

RETURN VALUE

Upon successful completion, the *tttk_message_reject*( ) function returns the status of the operation as one of the following **Tt_status** values:

TT_OK The operation completed successfully.

TT_ERR_NOMP
: The *ttsession* process is not running and the ToolTalk service cannot restart it.

TT_ERR_NOTHANDLER
: This application is not the handler for this message.

TT_ERR_POINTER
: The pointer passed does not point to an object of the correct type for this operation.

SEE ALSO

<Tt/tttk.h>, *tt_message_reject*( ), *tttk_message_fail*( ), *tttk_message_abandon*( ), *tttk_message_fail*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_op_string — map a ToolTalk op code to a string

SYNOPSIS

```
#include <Tt/tttk.h>

char *tttk_op_string(Tttk_op opcode);
```

DESCRIPTION

The *tttk_op_string*( ) function returns a string containing the op for *opcode*.

RETURN VALUE

Upon successful completion, the *tttk_op_string*( ) function returns a string that can be freed using *tt_free*( ); otherwise, it returns NULL.

APPLICATION USAGE

The distinctions in the **Tttk_op** enumerated type are for programmer convenience, and elements of **Tttk_op** do not necessarily map one-to-one with op strings, since ToolTalk allows ops to be overloaded. For example, TTME_EDIT and TTME_COMPOSE are overloaded on the same op (*Edit*), and the messages only vary by the **Tt_mode** of the first argument.

SEE ALSO

<Tt/tttk.h>, *tt_message_op*( ), *tt_free*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tttk_string_op — map a string to a ToolTalk op code

SYNOPSIS

```
#include <Tt/tttk.h>

Tttk_op tttk_string_op(const char *opstring);
```

DESCRIPTION

The *tttk_string_op*( ) function returns the **Tttk_op** named by *opstring*.

RETURN VALUE

Upon successful completion, the *tttk_string_op*( ) function a **Tttk_op** value; otherwise, it returns TTDT_OP_NONE.

APPLICATION USAGE

See *tttk_op_string*( ).

SEE ALSO

<Tt/tttk.h>, *tttk_op_string*( ), *tt_message_op*( ).

CHANGE HISTORY

First released in Issue 1.

6.3 Headers

This section describes the contents of headers used by the XCDE message service functions, macros and external variables.

Headers contain the definition of symbolic constants, common structures, preprocessor macros and defined types. Each function in Section 6.2 specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

NAME

Tt/tt_c.h — ToolTalk definitions

SYNOPSIS

```
#include <Tt/tt_c.h>
```

DESCRIPTION

The **<Tt/tt_c.h>** header includes **typedef**s for the following callback functions:

```
typedef Tt_filter_action (*Tt_filter_function)(const char *nodeid,
                                               void *context,
                                               void *accumulator);

typedef Tt_callback_action (*Tt_message_callback)(Tt_message m,
                                                  Tt_pattern p);
```

The header defines the TT_VERSION constant with the value 10200, indicating the version of the ToolTalk API.

The header defines the **Tt_status** enumeration data type, with the following members and specific values:

```
typedef enum tt_status {
        TT_OK                   = 0,
        TT_WRN_NOTFOUND         = 1,
        TT_WRN_STALE_OBJID      = 2,
        TT_WRN_STOPPED          = 3,
        TT_WRN_SAME_OBJID       = 4,
        TT_WRN_START_MESSAGE    = 5,
        TT_WRN_APPFIRST         = 512,
        TT_WRN_LAST             = 1024,
        TT_ERR_CLASS            = 1025,
        TT_ERR_DBAVAIL          = 1026,
        TT_ERR_DBEXIST          = 1027,
        TT_ERR_FILE             = 1028,
        TT_ERR_INVALID          = 1029,
        TT_ERR_MODE             = 1031,
        TT_ERR_ACCESS           = 1032,
        TT_ERR_NOMP             = 1033,
        TT_ERR_NOTHANDLER       = 1034,
        TT_ERR_NUM              = 1035,
        TT_ERR_OBJID            = 1036,
        TT_ERR_OP               = 1037,
        TT_ERR_OTYPE            = 1038,
        TT_ERR_ADDRESS          = 1039,
        TT_ERR_PATH             = 1040,
        TT_ERR_POINTER          = 1041,
        TT_ERR_PROCID           = 1042,
        TT_ERR_PROPLEN          = 1043,
        TT_ERR_PROPNAME         = 1044,
        TT_ERR_PTYPE            = 1045,
        TT_ERR_DISPOSITION      = 1046,
        TT_ERR_SCOPE            = 1047,
        TT_ERR_SESSION          = 1048,
        TT_ERR_VTYPE            = 1049,
        TT_ERR_NO_VALUE         = 1050,
```

```
TT_ERR_INTERNAL         = 1051,
TT_ERR_READONLY         = 1052,
TT_ERR_NO_MATCH         = 1053,
TT_ERR_UNIMP            = 1054,
TT_ERR_OVERFLOW         = 1055,
TT_ERR_PTYPE_START      = 1056,
TT_ERR_CATEGORY         = 1057,
TT_ERR_DBUPDATE         = 1058,
TT_ERR_DBFULL           = 1059,
TT_ERR_DBCONSIST        = 1060,
TT_ERR_STATE            = 1061,
TT_ERR_NOMEM            = 1062,
TT_ERR_SLOTNAME         = 1063,
TT_ERR_XDR              = 1064,
TT_ERR_NETFILE          = 1065,
TT_DESKTOP_             = 1100,
TT_DESKTOP_EPERM        = 1101,
TT_DESKTOP_ENOENT       = 1102,
TT_DESKTOP_EINTR        = 1104,
TT_DESKTOP_EIO          = 1105,
TT_DESKTOP_EAGAIN       = 1111,
TT_DESKTOP_ENOMEM       = 1112,
TT_DESKTOP_EACCES       = 1113,
TT_DESKTOP_EFAULT       = 1114,
TT_DESKTOP_EEXIST       = 1117,
TT_DESKTOP_ENODEV       = 1119,
TT_DESKTOP_ENOTDIR      = 1120,
TT_DESKTOP_EISDIR       = 1121,
TT_DESKTOP_EINVAL       = 1122,
TT_DESKTOP_ENFILE       = 1123,
TT_DESKTOP_EMFILE       = 1124,
TT_DESKTOP_ETXTBSY      = 1126,
TT_DESKTOP_EFBIG        = 1127,
TT_DESKTOP_ENOSPC       = 1128,
TT_DESKTOP_EROFS        = 1130,
TT_DESKTOP_EMLINK       = 1131,
TT_DESKTOP_EPIPE        = 1132,
TT_DESKTOP_ENOMSG       = 1135,
TT_DESKTOP_EDEADLK      = 1145,
TT_DESKTOP_ECANCELED    = 1147,
TT_DESKTOP_ENOTSUP      = 1148,
TT_DESKTOP_ENODATA      = 1161,
TT_DESKTOP_EPROTO       = 1171,
TT_DESKTOP_ENOTEMPTY    = 1193,
TT_DESKTOP_ETIMEDOUT    = 1245,
TT_DESKTOP_EALREADY     = 1249,
TT_DESKTOP_UNMODIFIED   = 1299,
TT_MEDIA_ERR_SIZE       = 1300,
TT_MEDIA_ERR_FORMAT     = 1301,
TT_ERR_APPFIRST         = 1536,
TT_ERR_LAST             = 2047,
TT_STATUS_LAST          = 2048
```

```
} Tt_status;
```

Note: Specific values are required because they can be communicated between ToolTalk clients on different platforms, usually via *tt_message_status_set*( ) and *tt_message_status*( ).

The header defines the following enumeration data types, with at least the following members:

Tt_filter_action
TT_FILTER_CONTINUE, TT_FILTER_LAST, TT_FILTER_STOP

Tt_callback_action
TT_CALLBACK_CONTINUE, TT_CALLBACK_LAST, TT_CALLBACK_PROCESSED

Tt_mode
TT_IN, TT_INOUT, TT_MODE_LAST, TT_MODE_UNDEFINED, TT_OUT

Tt_scope
TT_BOTH, TT_FILE, TT_FILE_IN_SESSION, TT_SCOPE_NONE, TT_SESSION

Tt_class
TT_CLASS_LAST, TT_CLASS_UNDEFINED, TT_NOTICE, TT_REQUEST

Tt_category
TT_CATEGORY_LAST, TT_CATEGORY_UNDEFINED, TT_HANDLE, TT_OBSERVE

Tt_address
TT_ADDRESS_LAST, TT_HANDLER, TT_OBJECT, TT_OTYPE, TT_PROCEDURE

Tt_disposition
TT_DISCARD, TT_QUEUE, TT_START

Tt_state
TT_CREATED, TT_FAILED, TT_HANDLED, TT_QUEUED, TT_REJECTED, TT_SENT, TT_STARTED, TT_STATE_LAST

The header defines the following as opaque data types: **Tt_message**, **Tt_pattern**.

The header declares the following as functions:

```
char *tt_X_session(const char *xdisplaystring);

Tt_status tt_bcontext_join(const char *slotname,
                           const unsigned char *value,
                           int length);

Tt_status tt_bcontext_quit(const char *slotname,
                           const unsigned char *value,
                           int length);

Tt_status tt_close(void);

Tt_status tt_context_join(const char *slotname,
                          const char *value);

Tt_status tt_context_quit(const char *slotname,
                          const char *value);

char *tt_default_file(void);
```

```
Tt_status tt_default_file_set(const char *docid);
char *tt_default_procid(void);
Tt_status tt_default_procid_set(const char *procid);
char *tt_default_ptype(void);
Tt_status tt_default_ptype_set(const char *ptid);
char *tt_default_session(void);
Tt_status tt_default_session_set(const char *sessid);
int tt_error_int(Tt_status ttrc);
void *tt_error_pointer(Tt_status ttrc);
int tt_fd(void);
Tt_status tt_file_copy(const char *oldfilepath,
                       const char *newfilepath);
Tt_status tt_file_destroy(const char *filepath);
Tt_status tt_file_join(const char *filepath);
Tt_status tt_file_move(const char *oldfilepath,
                       const char *newfilepath);
char *tt_file_netfile(const char *filename);
Tt_status tt_file_objects_query(const char *filepath,
                                Tt_filter_function filter,
                                void *context,
                                void *accumulator);
Tt_status tt_file_quit(const char *filepath);
void tt_free(caddr_t p);
char *tt_host_file_netfile(const char *host,
                           const char *filename);
char *tt_host_netfile_file(const char *host,
                           const char *netfilename);
Tt_status tt_icontext_join(const char *slotname, int value);
Tt_status tt_icontext_quit(const char *slotname, int value);
char *tt_initial_session(void);
Tt_status tt_int_error(int return_val);
int tt_is_err(Tt_status s);
caddr_t tt_malloc(size_t s);
int tt_mark(void);
Tt_status tt_message_accept(Tt_message m);
Tt_address tt_message_address(Tt_message m);
Tt_status tt_message_address_set(Tt_message m, Tt_address a);
```

```
Tt_status tt_message_arg_add(Tt_message m,
                             Tt_mode n,
                             const char *vtype,
                             const char *value);

Tt_status tt_message_arg_bval(Tt_message m,
                              int n,
                              unsigned char **value,
                              int *len);

Tt_status tt_message_arg_bval_set(Tt_message m,
                                  int n,
                                  const unsigned char *value,
                                  int len);

Tt_status tt_message_arg_ival(Tt_message m,
                              int n,
                              int *value);

Tt_status tt_message_arg_ival_set(Tt_message m,
                                  int n,
                                  int value);

Tt_mode tt_message_arg_mode(Tt_message m,
                            int n);

char *tt_message_arg_type(Tt_message m,
                          int n);

char *tt_message_arg_val(Tt_message m,
                         int n);

Tt_status tt_message_arg_val_set(Tt_message m,
                                 int n,
                                 const char *value);

Tt_status tt_message_arg_xval(Tt_message m,
                              int n,
                              xdrproc_t xdr_proc,
                              void **value);

Tt_status tt_message_arg_xval_set(Tt_message m,
                                  int n,
                                  xdrproc_t xdr_proc,
                                  void *value);

int tt_message_args_count(Tt_message m);

Tt_status tt_message_barg_add(Tt_message m,
                              Tt_mode n,
                              const char *vtype,
                              const unsigned char *value,
                              int len);

Tt_status tt_message_bcontext_set(Tt_message m,
                                  const char *slotname,
                                  const unsigned char *value,
                                  int length);
```

```
Tt_status tt_message_callback_add(Tt_message m,
                                  Tt_message_callback f);

Tt_class tt_message_class(Tt_message m);

Tt_status tt_message_class_set(Tt_message m,
                               Tt_class c);

Tt_status tt_message_context_bval(Tt_message m,
                                  const char *slotname,
                                  unsigned char **value,
                                  int *len);

Tt_status tt_message_context_ival(Tt_message m,
                                  const char *slotname,
                                  int *value);

Tt_status tt_message_context_set(Tt_message m,
                                 const char *slotname,
                                 const char *value);

char *tt_message_context_slotname(Tt_message m,
                                  int n);

char *tt_message_context_val(Tt_message m,
                             const char *slotname);

Tt_status tt_message_context_xval(Tt_message m,
                                  const char *slotname,
                                  xdrproc_t xdr_proc,
                                  void **value);

int tt_message_contexts_count(Tt_message m);

Tt_message tt_message_create(void);

Tt_message tt_message_create_super(Tt_message m);

Tt_status tt_message_destroy(Tt_message m);

Tt_disposition tt_message_disposition(Tt_message m);

Tt_status tt_message_disposition_set(Tt_message m,
                                     Tt_disposition r);

Tt_status tt_message_fail(Tt_message m);

char *tt_message_file(Tt_message m);

Tt_status tt_message_file_set(Tt_message m,
                              const char *file);

gid_t tt_message_gid(Tt_message m);

char *tt_message_handler(Tt_message m);

char *tt_message_handler_ptype(Tt_message m);

Tt_status tt_message_handler_ptype_set(Tt_message m,
                                       const char *ptid);

Tt_status tt_message_handler_set(Tt_message m,
                                 const char *procid);
```

```
Tt_status tt_message_iarg_add(Tt_message m,
                              Tt_mode n,
                              const char *vtype,
                              int value);

Tt_status tt_message_icontext_set(Tt_message m,
                                  const char *slotname,
                                  int value);

char *tt_message_id(Tt_message m);

char *tt_message_object(Tt_message m);

Tt_status tt_message_object_set(Tt_message m,
                                const char *objid);

char *tt_message_op(Tt_message m);

Tt_status tt_message_op_set(Tt_message m,
                            const char *opname);

int tt_message_opnum(Tt_message m);

char *tt_message_otype(Tt_message m);

Tt_status tt_message_otype_set(Tt_message m,
                               const char *otype);

Tt_pattern tt_message_pattern(Tt_message m);

char *tt_message_print(Tt_message *m);

Tt_message tt_message_receive(void);

Tt_status tt_message_reject(Tt_message m);

Tt_status tt_message_reply(Tt_message m);

Tt_scope tt_message_scope(Tt_message m);

Tt_status tt_message_scope_set(Tt_message m,
                               Tt_scope s);

Tt_status tt_message_send(Tt_message m);

Tt_status tt_message_send_on_exit(Tt_message m);

char *tt_message_sender(Tt_message m);

char *tt_message_sender_ptype(Tt_message m);

Tt_status tt_message_sender_ptype_set(Tt_message m,
                                      const char *ptid);

char *tt_message_session(Tt_message m);

Tt_status tt_message_session_set(Tt_message m,
                                 const char *sessid);

Tt_state tt_message_state(Tt_message m);

int tt_message_status(Tt_message m);

Tt_status tt_message_status_set(Tt_message m,
                                int status);
```

```
char *tt_message_status_string(Tt_message m);

Tt_status tt_message_status_string_set(Tt_message m,
                                       const char *status_str);

uid_t tt_message_uid(Tt_message m);

void *tt_message_user(Tt_message m,
                      int key);

Tt_status tt_message_user_set(Tt_message m,
                              int key,
                              void *v);

Tt_status tt_message_xarg_add(Tt_message m,
                              Tt_mode n,
                              const char *vtype,
                              xdrproc_t xdr_proc,
                              void *value);

Tt_status tt_message_xcontext_join(const char *slotname,
                                   xdrproc_t xdr_proc,
                                   void *value);

Tt_status tt_message_xcontext_set(Tt_message m,
                                  const char *slotname,
                                  xdrproc_t xdr_proc,
                                  void *value);

char *tt_netfile_file(const char *netfilename);

int tt_objid_equal(const char *objid1,
                   const char *objid2);

char *tt_objid_objkey(const char *objid);

Tt_message tt_onotice_create(const char *objid,
                             const char *op);

char *tt_open(void);

Tt_message tt_orequest_create(const char *objid,
                              const char *op);

char *tt_otype_base(const char *otype);

char *tt_otype_derived(const char *otype,
                       int i);

int tt_otype_deriveds_count(const char *otype);

Tt_mode tt_otype_hsig_arg_mode(const char *otype,
                               int sig,
                               int arg);

char *tt_otype_hsig_arg_type(const char *otype,
                             int sig,
                             int arg);

int tt_otype_hsig_args_count(const char *otype,
                             int sig);
```

```
int tt_otype_hsig_count(const char *otype);

char *tt_otype_hsig_op(const char *otype,
                       int sig);

int tt_otype_is_derived(const char *derivedotype,
                        const char *baseotype);

Tt_status tt_otype_opnum_callback_add(const char *otid,
                                      int opnum,
                                      Tt_message_callback f);

Tt_mode tt_otype_osig_arg_mode(const char *otype,
                               int sig,
                               int arg);

char *tt_otype_osig_arg_type(const char *otype,
                             int sig,
                             int arg);

int tt_otype_osig_args_count(const char *otype,
                             int sig);

int tt_otype_osig_count(const char*otype);

char *tt_otype_osig_op(const char *otype,
                       int sig);

Tt_status tt_pattern_address_add(Tt_pattern p,
                                 Tt_address d);

Tt_status tt_pattern_arg_add(Tt_pattern p,
                             Tt_mode n,
                             const char *vtype,
                             const char *value);

Tt_status tt_pattern_barg_add(Tt_pattern m,
                              Tt_mode n,
                              const char *vtype,
                              const unsigned char *value,
                              int len);

Tt_status tt_pattern_bcontext_add(Tt_pattern p,
                                  const char *slotname,
                                  const unsigned char *value,
                                  int length);

Tt_status tt_pattern_callback_add(Tt_pattern m,
                                  Tt_message_callback f);

Tt_category tt_pattern_category(Tt_pattern p);

Tt_status tt_pattern_category_set(Tt_pattern p,
                                  Tt_category c);

Tt_status tt_pattern_class_add(Tt_pattern p,
                               Tt_class c);

Tt_status tt_pattern_context_add(Tt_pattern p,
                                 const char *slotname,
                                 const char *value);
```

```
Tt_pattern tt_pattern_create(void);

Tt_status tt_pattern_destroy(Tt_pattern p);

Tt_status tt_pattern_disposition_add(Tt_pattern p,
                                     Tt_disposition r);

Tt_status tt_pattern_file_add(Tt_pattern p,
                              const char *file);

Tt_status tt_pattern_iarg_add(Tt_pattern m,
                              Tt_mode n,
                              const char *vtype,
                              int value);

Tt_status tt_pattern_icontext_add(Tt_pattern p,
                                  const char *slotname,
                                  int value);

Tt_status tt_pattern_object_add(Tt_pattern p,
                                const char *objid);

Tt_status tt_pattern_op_add(Tt_pattern p,
                            const char *opname);

Tt_status tt_pattern_opnum_add(Tt_pattern p,
                               int opnum);

Tt_status tt_pattern_otype_add(Tt_pattern p,
                               const char *otype);

char *tt_pattern_print(Tt_pattern *p);

Tt_status tt_pattern_register(Tt_pattern p);

Tt_status tt_pattern_scope_add(Tt_pattern p,
                               Tt_scope s);

Tt_status tt_pattern_sender_add(Tt_pattern p,
                                const char *procid);

Tt_status tt_pattern_sender_ptype_add(Tt_pattern p,
                                      const char *ptid);

Tt_status tt_pattern_session_add(Tt_pattern p,
                                 const char *sessid);

Tt_status tt_pattern_state_add(Tt_pattern p,
                               Tt_state s);

Tt_status tt_pattern_unregister(Tt_pattern p);

void *tt_pattern_user(Tt_pattern p,
                      int key);

Tt_status tt_pattern_user_set(Tt_pattern p,
                              int key,
                              void *v);

Tt_status tt_pattern_xarg_add(Tt_pattern m,
                              Tt_mode n,
                              const char *vtype,
```

```
                                xdrproc_t xdr_proc,
                                void *value);

Tt_status tt_pattern_xcontext_add(Tt_pattern p,
                                    const char *slotname,
                                    xdrproc_t xdr_proc,
                                    void *value);

Tt_message tt_pnotice_create(Tt_scope scope,
                               const char *op);

Tt_status tt_pointer_error(void *pointer);

Tt_message tt_prequest_create(Tt_scope scope,
                                const char *op);

Tt_status tt_ptr_error(pointer);

Tt_status tt_ptype_declare(const char *ptid);

Tt_status tt_ptype_exists(const char *ptid);

Tt_status tt_ptype_opnum_callback_add(const char *ptid,
                                      int opnum,
                                      Tt_message_callback f);

Tt_status tt_ptype_undeclare(const char *ptid);

void tt_release(int mark);

Tt_status tt_session_bprop(const char *sessid,
                           const char *propname,
                           int i,
                           unsigned char **value,
                           int *length);

Tt_status tt_session_bprop_add(const char *sessid,
                               const char *propname,
                               const unsigned char *value,
                               int length);

Tt_status tt_session_bprop_set(const char *sessid,
                               const char *propname,
                               const unsigned char *value,
                               int length);

Tt_status tt_session_join(const char *sessid);

char *tt_session_prop(const char *sessid,
                      const char *propname,
                      int i);

Tt_status tt_session_prop_add(const char *sessid,
                              const char *propname,
                              const char *value);

int tt_session_prop_count(const char *sessid,
                          const char *propname);

Tt_status tt_session_prop_set(const char *sessid,
                              const char *propname,
```

```
                                const char *value);

char *tt_session_propname(const char *sessid,
                          int n);

int tt_session_propnames_count(const char *sessid);

Tt_status tt_session_quit(const char *sessid);

Tt_status tt_session_types_load(const char *session,
                                const char *filename);

Tt_status tt_spec_bprop(const char *objid,
                        const char *propname,
                        int i,
                        unsigned char **value,
                        int *length);

Tt_status tt_spec_bprop_add(const char *objid,
                            const char *propname,
                            const unsigned char *value,
                            int length);

Tt_status tt_spec_bprop_set(const char *objid,
                            const char *propname,
                            const unsigned char *value,
                            int length);

char *tt_spec_create(const char *filepath);

Tt_status tt_spec_destroy(const char *objid);

char *tt_spec_file(const char *objid);

char *tt_spec_move(const char *objid,
                   const char *newfilepath);

char *tt_spec_prop(const char *objid,
                   const char *propname,
                   int i);

Tt_status tt_spec_prop_add(const char *objid,
                           const char *propname,
                           const char *value);

int tt_spec_prop_count(const char *objid,
                       const char *propname);

Tt_status tt_spec_prop_set(const char *objid,
                           const char *propname,
                           const char *value);

char *tt_spec_propname(const char *objid,
                       int n);

int tt_spec_propnames_count(const char *objid);

char *tt_spec_type(const char *objid);

Tt_status tt_spec_type_set(const char *objid,
                           const char *otid);
```

```
Tt_status tt_spec_write(const char *objid);
char *tt_status_message(Tt_status ttrc);
int tt_trace_control(int onoff);
Tt_status tt_xcontext_quit(const char *slotname,
                           xdrproc_t xdr_proc,
                           void *value);
```

CHANGE HISTORY

First released in Issue 1.

NAME

Tt/tttk.h — ToolTalk definitions

SYNOPSIS

```
#include <Tt/tttk.h>
```

DESCRIPTION

The **<Tt/tttk.h>** header defines the following enumeration data type, with at least the following members:

Tttk_op

TTDT_CREATED, TTDT_DELETED, TTDT_DO_COMMAND, TTDT_GET_ENVIRONMENT, TTDT_GET_GEOMETRY, TTDT_GET_ICONIFIED, TTDT_GET_LOCALE, TTDT_GET_MAPPED, TTDT_GET_MODIFIED, TTDT_GET_SITUATION, TTDT_GET_STATUS, TTDT_GET_SYSINFO, TTDT_GET_XINFO, TTDT_LOWER, TTDT_MODIFIED, TTDT_MOVED, TTDT_OP_LAST, TTDT_OP_NONE, TTDT_PAUSE, TTDT_QUIT, TTDT_RAISE, TTDT_RESUME, TTDT_REVERT, TTDT_REVERTED, TTDT_SAVE, TTDT_SAVED, TTDT_SET_ENVIRONMENT, TTDT_SET_GEOMETRY, TTDT_SET_ICONIFIED, TTDT_SET_LOCALE, TTDT_SET_MAPPED, TTDT_SET_SITUATION, TTDT_SET_XINFO, TTDT_SIGNAL, TTDT_STARTED, TTDT_STATUS, TTDT_STOPPED, TTME_ABSTRACT, TTME_COMPOSE, TTME_DEPOSIT, TTME_DISPLAY, TTME_EDIT, TTME_INTERPRET, TTME_MAIL, TTME_MAIL_COMPOSE, TTME_MAIL_EDIT, TTME_PRINT, TTME_TRANSLATE

The header declares the following global string constants for some standard vtypes:

```
extern const char  *Tttk_boolean:
extern const char  *Tttk_file:
extern const char  *Tttk_height:
extern const char  *Tttk_integer:
extern const char  *Tttk_message_id:
extern const char  *Tttk_string:
extern const char  *Tttk_title:
extern const char  *Tttk_width:
extern const char  *Tttk_xoffset:
extern const char  *Tttk_yoffset:
```

The header declares the following as functions:

```
int ttdt_Get_Modified(Tt_message context,
                      const char *pathname,
                      Tt_scope the_scope,
                      XtAppContext app2run,
                      int ms_timeout);

Tt_status ttdt_Revert(Tt_message context,
                      const char *pathname,
                      Tt_scope the_scope,
                      XtAppContext app2run,
                      int ms_timeout);

Tt_status ttdt_Save(Tt_message context,
                    const char *pathname,
                    Tt_scope the_scope,
                    XtAppContext app2run,
```

```
                    int ms_timeout);

Tt_status ttdt_close(const char *procid,
                     const char *new_procid,
                     int sendStopped);

Tt_status ttdt_file_event(Tt_message context,
                          Tttk_op event,
                          Tt_pattern *patterns,
                          int send);

Tt_pattern *ttdt_file_join(const char *pathname,
                           Tt_scope the_scope,
                           int join,
                           Ttdt_file_cb cb,
                           void *clientdata);

Tt_message ttdt_file_notice(Tt_message context,
                            Tttk_op op,
                            Tt_scope scope,
                            const char *pathname,
                            int send_and_destroy);

Tt_status ttdt_file_quit(Tt_pattern *patterns,
                         int quit);

Tt_message ttdt_file_request(Tt_message context,
                             Tttk_op op,
                             Tt_scope scope,
                             const char *pathname,
                             Ttdt_file_cb cb,
                             void *client_data,
                             int send_and_destroy);

Tt_pattern *ttdt_message_accept(Tt_message contract,
                                Ttdt_contract_cb cb,
                                Widget shell,
                                void *clientdata,
                                int accept,
                                int sendStatus);

char *ttdt_open(int *ttfd,
                const char *toolname,
                const char *vendor,
                const char *version,
                int sendStarted);

Tt_status ttdt_sender_imprint_on(const char *handler,
                                 Tt_message contract,
                                 char **display,
                                 int *width,
                                 int *height,
                                 int *xoffset,
                                 int *yoffset,
                                 XtAppContext app2run,
                                 int ms_timeout);
```

```
Tt_pattern *ttdt_session_join(const char *sessid,
                              Ttdt_contract_cb cb,
                              Widget shell,
                              void *clientdata,
                              int join);

Tt_status ttdt_session_quit(const char *sessid,
                            Tt_pattern *sess_pats,
                            int quit);

Tt_pattern *ttdt_subcontract_manage(Tt_message subcontract,
                                    Ttdt_contract_cb cb,
                                    Widget shell,
                                    void *clientdata);

Tt_status ttmedia_Deposit(Tt_message load_contract,
                          const char *buffer_id,
                          const char *media_type,
                          const unsigned char *new_contents,
                          int new_len,
                          const char *pathname,
                          XtAppContext app2run,
                          int ms_timeout);

Tt_message ttmedia_load(Tt_message context,
                        Ttmedia_load_msg_cb cb,
                        void *clientdata,
                        Tttk_op op,
                        const char *media_type,
                        const unsigned char *contents,
                        int len,
                        const char *file,
                        const char *docname,
                        int send);

Tt_message ttmedia_load_reply(Tt_message contract,
                              const unsigned char *new_contents,
                              int new_len,
                              int reply_and_destroy);

Tt_status ttmedia_ptype_declare(const char *ptype,
                                int base_opnum,
                                Ttmedia_load_pat_cb cb,
                                void *clientdata,
                                int declare);

void tttk_Xt_input_handler(XtPointer procid,
                           int *source,
                           XtInputId *id);

Tt_status tttk_block_while(XtAppContext app2run,
                           const int *blocked,
                           int ms_timeout);

Tt_status tttk_message_abandon(Tt_message msg);
```

```
Tt_message tttk_message_create(Tt_message context,
                               Tt_class the_class,
                               Tt_scope the_scope,
                               const char *handler,
                               const char *op,
                               Tt_message_callback callback);

Tt_status tttk_message_destroy(Tt_message msg);

Tt_status tttk_message_fail(Tt_message msg,
                            Tt_status status,
                            const char *status_string,
                            int destroy);

Tt_status tttk_message_reject(Tt_message msg,
                              Tt_status status,
                              const char *status_string,
                              int destroy);

char *tttk_op_string(Tttk_op opcode);

Tttk_op tttk_string_op(const char *opstring);
```

CHANGE HISTORY

First released in Issue 1.

6.4 Command-Line Interfaces

This section defines the utilities that provide XCDE message services.

NAME

tt_type_comp — compile ToolTalk otypes and ptypes

SYNOPSIS

```
tt_type_comp [–mMs] source_file
tt_type_comp –r [–s] type ...
tt_type_comp –p | –O | –P [–s]
tt_type_comp –p | –O | –P [–s] source_file
tt_type_comp –x [–s] [–o compiled_file] source_file
tt_type_comp [–hv]
```

DESCRIPTION

The *tt_type_comp* utility processes otypes and ptypes. The default action of *tt_type_comp* is to compile types from source form into compiled form and then merge the compiled types into the standard ToolTalk types databases. The *tt_type_comp* utility preprocesses the source types with *cpp*, and can optionally write out the compiled types instead of merging them into the standard databases. The *tt_type_comp* utility can also remove types from the standard databases or write out the contents of these databases.

OPTIONS

The *tt_type_comp* utility supports the X/Open Utility Syntax Guidelines. The following options are available:

–h Write a help message for invoking *tt_type_comp* and then exit.

–m Merge types into the specified database, updating any existing type with the new definition given. This is the default action. The specified database is the first element from the *TTPATH* environment variable, or **$HOME/.tt/types.xdr** if *TTPATH* is NULL or not set. If *TTPATH* is NULL or not set, it is considered to be:

```
$HOME/.tt/types.xdr:\
/etc/tt/types.xdr:\
/usr/dt/appconfig/tttypes/types.xdr
```

–M Merge types into the specified database (see **–m**), but only if they do not already exist in that database.

–O Write the names of all otypes read.

–p Write the ToolTalk types read in a source format suitable for recompilation with *tt_type_comp*.

–P Write the names of all ptypes read.

–o *compiled_file*
Write the compiled types into the specified file, or to standard output if *compiled_file* is –.

–r Remove the given ptypes or otypes from the specified database, as indicated by the *type* operands.

–s Silent mode. Write nothing to standard output.

–v Write the version number of *tt_type_comp* and then exit.

–x Compile source types into a compiled types file, instead of merging them into the standard types databases.

OPERANDS

The following operands are supported:

source_file
: A pathname of a text file containing ToolTalk source code. If *source_file* is –, standard input is used.

type
: A name of a type to be removed by the –**r** option.

STDIN

The standard input is used only if a *source_file* operand is –.

INPUT FILES

The input file named by *source_file* is a text file containing ToolTalk source code, which must conform to the format described in Section 6.5 on page 358.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tt_type_comp*:

LANG
: Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility behaves as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_MESSAGES
: Determine the locale that is used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

NLSPATH
: Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TTPATH
: A colon-separated list of directories that tells the ToolTalk service where to find the ToolTalk types databases.

RESOURCES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the –**h** option is used, *tt_type_comp* writes to standard output a help message in an unspecified format.

When the –**o** option is used, *tt_type_comp* writes to standard output a listing of all otypes read.

When the –**p** option is used, *tt_type_comp* writes to standard output a listing of all the ToolTalk types read, in a source format suitable for recompilation with *tt_type_comp*.

When the –**P** option is used, *tt_type_comp* writes to standard output a listing of all ptypes read.

When the –**v** option is used, *tt_type_comp* writes to standard output a version number in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

When the **–x** or **–d user** option is used, *tt_type_comp* writes the compiled types in an unspecified format into a user-specified file. Otherwise, it writes the compiled types into the databases described under **–d**.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.

1 Usage; *tt_type_comp* was given invalid command line options.

2 A syntax error was found in the source types given to *tt_type_comp*.

3 System error; *tt_type_comp* was interrupted by SIGINT, or encountered some system or internal error.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

SEE ALSO

ttsession, *cpp*.

CHANGE HISTORY

First released in Issue 1.

NAME

ttcp — copy files and inform the ToolTalk service

SYNOPSIS

```
ttcp [–pL] filename1 filename2
ttcp –r [–pL] directory1 directory2
ttcp [–prL] filename ... directory
ttcp –h | –v
```

DESCRIPTION

The *ttcp* utility invokes the *cp* utility to copy files and directories, and informs ToolTalk about its actions so that the ToolTalk objects associated with those files and directories can also be copied.

OPTIONS

The *ttcp* utility supports the X/Open Utility Syntax Guidelines. The following options are available:

–h Write a help message for invoking *ttcp* and then exit.

–L Copy the ToolTalk objects of the files, but do not invoke *cp* to copy the actual files.

–p Preserve. Invoke *cp* with the **–p** option, which duplicates not only the contents of the original files or directories, but also the modification time and permission modes. The modification times of ToolTalk objects are preserved only if the invoking process has appropriate privileges.

–r Recursively copy the ToolTalk objects of any directories named, along with their files (including any subdirectories and their files), and pass the **–r** option to *cp*.

–v Write the version number of *ttcp* and then exit.

It is unspecified whether the **–f**, **–i** or **–R** options to *cp* are supported.

OPERANDS

The following operands are supported:

filename
filename1
A pathname of a file to be copied.

filename2
A pathname of an existing or nonexisting file, used for the output when a single file is copied.

directory
directory2
A pathname of a directory to contain the copied files.

directory1
A pathname of a file hierarchy to be copied with **–r**.

STDIN

Not used.

INPUT FILES

The input files specified as operands can be of any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ttcp*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility behaves as if none of the variables had been defined.

LC_ALL If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_MESSAGES Determine the locale that is used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

NLSPATH Determine the location of message catalogues for the processing of *LC_MESSAGES*.

RESOURCES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **–h** option is used, *ttcp* writes to standard output a help message in an unspecified format.

When the **–v** option is used, *ttcp* writes to standard output a version number in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output files can be of any type.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All files and ToolTalk objects were copied successfully.

>0 An error occurred or the invoked *cp* command exited with a non-zero value.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

SEE ALSO

cp in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**; *ttmv*, *tttar*, *ttsession*.

CHANGE HISTORY

First released in Issue 1.

NAME

ttmv — move or rename files and inform the ToolTalk service

SYNOPSIS

```
ttmv [–fL] pathname1 pathname2
ttmv [–fL] pathname . . .  directory
ttmv –h | –v
```

DESCRIPTION

The *ttmv* utility invokes *mv* to move files and directories around in the file system and informs ToolTalk about its actions so that the ToolTalk objects associated with those files and directories can also be moved.

The *ttmv* utility moves the ToolTalk objects before it moves the files and does not check whether the file-moving operation will succeed before performing the object-moving operation.

OPTIONS

The *ttmv* utility supports the X/Open Utility Syntax Guidelines. The following options are available:

–f Force. Do not report any errors, and pass the **–f** option to *mv*.

–h Write a help message for invoking *ttmv* and then exit.

–L Move the ToolTalk objects of the files, but do not invoke *mv* to move the actual files.

–v Write the version number of *ttmv* and then exit.

It is unspecified whether the **–i** option to *mv* is supported.

OPERANDS

The following operands are supported:

pathname1
A pathname of a file to be moved.

pathname2
A pathname of an existing or nonexisting file, used for the output when a single file is moved.

directory
A pathname of a directory to contain the moved files.

STDIN

Not used.

INPUT FILES

The input files specified as operands can be of any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ttmv*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility behaves as if none of the variables had been defined.

LC_ALL If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_MESSAGES Determine the locale that is used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

NLSPATH Determine the location of message catalogues for the processing of *LC_MESSAGES*.

RESOURCES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **–h** option is used, *ttmv* writes to standard output a help message in an unspecified format.

When the **–v** option is used, *ttmv* writes to standard output a version number in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output files can be of any type.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All files and ToolTalk objects were moved successfully.

>0 An error occurred or the invoked *mv* command exited with a non-zero value.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

SEE ALSO

mv in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**; *ttsession*.

CHANGE HISTORY

First released in Issue 1.

NAME

ttrm — remove files or directories and inform the ToolTalk service

SYNOPSIS

```
ttrm [–frL] pathname ...
ttrm –h | –v
```

DESCRIPTION

The *ttrm* utility invokes *rm* to remove files and directories and informs ToolTalk about its actions so that the ToolTalk objects associated with the deleted files and directories can also be deleted.

The *ttrm* utility removes the ToolTalk objects before it removes the files and does not check whether the file-removing operation will succeed before performing the object-removing operation.

OPTIONS

The *ttrm* utility supports the X/Open Utility Syntax Guidelines. The following options are available:

–f Force. Do not report any errors, and pass the **–f** option to *rm*.

–h Write a help message for invoking *ttrm* and then exit.

–L Remove the ToolTalk objects of the files or directories, but do not invoke *rm* to remove the actual files or directories.

–r Recursively remove the ToolTalk objects of any directories named, along with their files (including any subdirectories and their files), and pass the **–r** option to *rm*.

–v Write the version number of *ttrm* and then exit.

It is unspecified whether the **–i** or **–R** options to *rm* are supported.

OPERANDS

The following operand is supported:

pathname
A pathname of a file to be removed.

STDIN

Not used.

INPUT FILES

The input files specified as operands can be of any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ttrm*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility behaves as if none of the variables had been defined.

LC_ALL If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_MESSAGES Determine the locale that is used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

NLSPATH Determine the location of message catalogues for the processing of *LC_MESSAGES*.

RESOURCES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **–h** option is used, *ttrm* writes to standard output a help message in an unspecified format.

When the **–v** option is used, *ttrm* writes to standard output a version number in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All files and ToolTalk objects were removed successfully.

>0 An error occurred or the invoked *rm* command exited with a non-zero value.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

SEE ALSO

rm in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**; *ttrmdir*, *ttsession*.

CHANGE HISTORY

First released in Issue 1.

NAME

ttrmdir — remove empty directories and inform the ToolTalk service

SYNOPSIS

```
ttrmdir [–L] directory ...
ttrmdir –h | –v
```

DESCRIPTION

The *ttrmdir* utility invokes *rmdir* to remove empty directories and informs ToolTalk about its actions so that the ToolTalk objects associated with the deleted directories can also be deleted.

The *ttrmdir* utility removes the ToolTalk objects before it removes the directories and does not check whether a directory is empty or whether the directory-removing operation will succeed before performing the object-removing operation.

OPTIONS

The *ttrmdir* utility supports the X/Open Utility Syntax Guidelines. The following options are available:

–h Write a help message for invoking *ttrmdir* and then exit.

–L Remove the ToolTalk objects of the directories, but do not invoke *rmdir* to remove the actual directories.

–v Write the version number of *ttrmdir* and then exit.

It is unspecified whether the **–p** option to *cp* is supported.

OPERANDS

The following operand is supported:

directory
A pathname of an empty directory to be removed.

STDIN

Not used.

INPUT FILES

The input files specified as operands can be of any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ttrmdir*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility behaves as if none of the variables had been defined.

LC_ALL If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_MESSAGES Determine the locale that is used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

NLSPATH Determine the location of message catalogues for the processing of *LC_MESSAGES*.

RESOURCES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **–h** option is used, *ttrmdir* writes to standard output a help message in an unspecified format.

When the **–v** option is used, *ttrmdir* writes to standard output a version number in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All directories and ToolTalk objects were removed successfully.

>0 An error occurred or the invoked *rmdir* command exited with a non-zero value.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The definition of an empty directory is one that contains, at most, directory entries for dot and dot-dot.

EXAMPLES

None.

SEE ALSO

rmdir in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**; *ttrm*, *ttsession*.

CHANGE HISTORY

First released in Issue 1.

NAME

ttsession — the ToolTalk message server

SYNOPSIS

`ttsession [–hNpsStv] [–E|–X] [–a` *`level`*`] [–d` *`display`*`] [–c [`*`command`*`]]`

DESCRIPTION

The *ttsession* utility is the ToolTalk message server. This background process must be running before any messages can be sent or received. Each message server defines a *session*.

The message server has no user interface and typically runs in the background, started either by the user's **.xinitrc** file or automatically by any program that needs to send or receive a message.

OPTIONS

The *ttsession* utility supports the X/Open Utility Syntax Guidelines, except that the **–c** option has an optional option-argument, which treats all of the following command-line arguments as a string to be passed to another shell invocation. The following options are available:

–a *level*
Set the server authentication level. The following *level* string values are supported:

unix The sender and receiver must have the same user ID.

des The underlying RPC calls use AUTH_DES.

–c [*command***]**
Start a process tree session and run the given command. The *ttsession* utility sets the environment variable *TT_SESSION* to the name of this session. Any process started with this variable in the environment defaults to being in this session. If *command* is omitted, *ttsession* invokes the shell named by the *SHELL* environment variable. Everything after **–c** on the command line is used as the command to be executed.

–d *display*
Specify an X Windows display. The ToolTalk session will consist of those applications displaying on the named display. The default display is identified by the *DISPLAY* environment variable.

–E Read in the types from the Classing Engine database. If neither **–E** nor **–X** is given, **–X** is assumed.

–h Write a help message to standard error that describes the command syntax of *ttsession*, and exit.

–N Maximise the number of clients allowed to connect to (in other words, open procids in) this session by attempting to raise the limit of open file descriptors. The precise number of clients is system-dependent; on some systems this option may have no effect.

–p Write the name of a new process tree session to standard output, and then fork a background instance of *ttsession* to manage this new session.

–s Silent. Do not write any warning messages to standard error.

–S Do not fork a background instance to manage the *ttsession* session.

–t Turn on trace mode. See **ASYNCHRONOUS EVENTS** for how to turn tracing on and off during execution. Tracing displays the state of a message when it is first seen by *ttsession*. The lifetime of the message is then shown by showing the result of matching the message against type signatures (dispatch stage) and then showing the result of matching the message against any registered message patterns

(delivery stage). Any attempt to send the message to a given process is also shown together with the success of that attempt.

–v Write the version number to standard output and exit.

–X Read in the types from the following XDR format databases:

```
$HOME/.tt/types.xdr
<implementation-specific system and network databases>
/usr/dt/appconfig/tttypes/types.xdr
```

The databases are listed order of decreasing precedence. Entries in **$HOME/.tt/types.xdr** override any like entries in the databases lower in the list, and so forth.

These locations can be overridden by setting the *TTPATH* environment variable. See **ENVIRONMENT VARIABLES**.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

The XDR format databases listed by the **–X** option are serialised ToolTalk data structures of an unspecified format, except that it is the same as the format of *tt_type_comp* output files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ttsession*:

DISPLAY If *TT_SESSION* is not set and *DISPLAY* is set, then the value of *DISPLAY* will be used by all ToolTalk clients to identify the *ttsession* process serving their X display. If no such process is running, the ToolTalk service will auto-start one.

If *ttsession* is run with the **–d** option and *DISPLAY* is not set, *ttsession* sets *DISPLAY* to be the value of the **–d** option for itself and all processes it forks. This helps ToolTalk clients to find the right X display when they are auto-started by *ttsession*.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility behaves as if none of the variables had been defined.

LC_ALL If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_MESSAGES Determine the locale that is used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

NLSPATH Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TT_ARG_TRACE_WIDTH
Specify the number of bytes of argument and context values to write when in trace mode. The default is to print the first 40 bytes.

TTPATH A colon-separated list of directories that tells ToolTalk where to find the ToolTalk types databases. The format of this variable is:

userDir[:*systemDir*[:*networkDir*]]

TTSESSION_CMD
Specify the shell command to be used by all ToolTalk clients for auto-starting *ttsession*.

The *ttsession* utility creates the following variable when it invokes another process:

TT_FILE When *ttsession* invokes a tool to receive a message, it copies the file attribute (if any) of the message into this variable, formatted in the same manner as returned by the *tt_message_file*( ) function.

TT_SESSION The *ttsession* utility uses this variable to communicate its session ID to the tools that it starts. The format of the variable is implementation specific. If this variable is set, the ToolTalk client library uses its value as the default session ID.

TT_TOKEN Inform the ToolTalk client library that it has been invoked by *ttsession*, so that the client can confirm to *ttsession* that it started successfully. The format of the variable is implementation specific.

A tool started by *ttsession* must ensure that the *TT_SESSION* and *TT_TOKEN* are present in the environment of any processes it invokes.

RESOURCES
None.

ASYNCHRONOUS EVENTS
The *ttsession* utility reacts to two signals. If it receives the SIGUSR1 signal, it toggles trace mode on or off (see the **–t** option). If it receives the SIGUSR2 signal, it rereads the types file. The *ttsession* utility takes the standard action for all other signals.

STDOUT
When the **–v** option is used, *ttsession* writes the version number in an unspecified format. When **–p** is used, *ttsession* writes the name of a new process tree session.

STDERR
Used only for diagnostic messages and the help message written by the **–h** option.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
When the **–c** child process exits, *ttsession* exits with the status of the exited child. Otherwise, the following exit values are returned:

0 Normal termination. Without the **–c** or **–S** options, a zero exit status means *ttsession* has successfully forked an instance of itself that has begun serving the session.

1 Abnormal termination. The *ttsession* utility was given invalid command line options, was interrupted by SIGINT, or encountered some internal error.

2 Collision. Another *ttsession* was found to be serving the session already.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since everything after **–c** on the command line is used as the command to be executed, **–c** should be the last option.

Tracing is helpful for seeing how messages are dispatched and delivered, but the output may be voluminous.

EXAMPLES

None.

SEE ALSO

tt_type_comp, *tt_message_file*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

tttar — process files and ToolTalk objects in an archive

SYNOPSIS

`tttar c | t | x [EfhpSv] [`*`tarfile`*` ]` *`pathname`* `...`

`tttar c | t | xfL [EhpRSv]` *`tttarfile`* `[[–rename` *`oldname newname`*`] ...]` *`pathname`* `...`

`tttar –h | –help`

`tttar –v`

DESCRIPTION

The *tttar* utility has two fundamentally different modes.

- Without the **L** function modifier, *tttar* acts as a ToolTalk-aware wrapper for *tar*, archiving (or extracting) multiple files and their ToolTalk objects onto (or from) a single archive, called a *tarfile*.
- With the **L** function modifier, *tttar* does not invoke **tar** to archive actual files, but instead archives (or extracts) only ToolTalk objects onto (or from) a single archive, called a *tttarfile*. Since without the **L** function modifier *tttar* acts like an ToolTalk-aware *tar*, the description below is phrased as if the **L** function modifier is in effect. That is, the text refers to *tttarfiles* instead of *tarfiles*, and it describes archiving and de-archiving only ‘‘the ToolTalk objects of the named files’’ rather than archiving and de-archiving both ‘‘the named files and their ToolTalk objects.’’

The actions of *tttar* are controlled by the first argument, the *key*, a string of characters containing exactly one function letter from the set **ctx**, and one or more of the optional function modifiers listed under **OPERANDS**. Other arguments to *tttar* are file or directory names that specify which files to archive or extract ToolTalk objects for. By default, the appearance of a directory name refers recursively to the files and subdirectories of that directory.

A file does not have to exist for a ToolTalk object to be associated with its pathname. When *tttar* descends into a directory, it does not attempt to archive the objects associated with any files that do not exist in the directory.

When extracting from a **tar** archive that is given to *tttar* either on magnetic tape or on the standard input, the current working directory must be writable, so that the *tttarfile* can be placed there temporarily.

OPTIONS

The *tttar* utility supports the X/Open Utility Syntax Guidelines, except that the –**help** and –**rename** options are full words, which cannot be combined with the other options, and –**rename** can only be used after the first operand, *tttarfile*. The following options are available:

–**h**

–**help** Write a help message for invoking *tttar* and then exit.

–**rename** *oldname newname*

Interpret the next two arguments as an *oldname* and a *newname*, respectively, and rename any entry archived as *oldname* to *newname*. If *oldname* is a directory, then *tttar* recursively renames the entries as well. If more than one –**rename** option applies to an entry (because of one or more parent directories being renamed), the most specific –**rename** option applies.

–**v** Write the version number of *tttar* and then exit.

OPERANDS

The following operands are supported:

key The *key* operand consists of a function letter followed immediately by zero or more modifying letters.

The function letter is one of the following:

c Create a new archive and write the ToolTalk objects of the named files onto it.

t Write to standard output the names of all the files in the archive.

x Extract the ToolTalk objects of the named files from the archive. If a named file matches a directory with contents in the archive, this directory is (recursively) extracted. The owner and modification time of the ToolTalk objects are restored (if possible). If no *filename* arguments are given, the ToolTalk objects of all files named in the archive are extracted.

The following characters can be appended to the function letter. Appending the same character more than once produces undefined results.

f Use the next argument as the name of the *tttarfile*. If *tttarfile* is given as '–', *tttar* writes to the standard output or reads from the standard input, whichever is appropriate.

h Follow symbolic links as if they were normal files or directories. Normally, *tttar* does not follow symbolic links.

p Preserve. Restore the named files to their original modes, ignoring the present *umask* value (see *umask*( )).

L Do not invoke *tar*.

R Do not recurse into directories.

v Verbose. Write to standard error the name of each file processed, preceded by a string indicating the operation being performed, as follows:

Key Letter	String
c	"a "
x	"x "

The file name may be followed by additional information, such as the size of the file in the archive or file system, in an unspecified format. When used with the **t** function letter, **v** writes to standard output more information about the archive entries than just the name.

It is unspecified whether the following functions and modifiers are supported:

- The **r** and **u** function letters of *tar*, for incrementally updating an archive.
- The **X** and **F** function modifiers and the –**I** option of *tar*, for including or excluding files from being archived based on SCCS status or being listed in a special file.
- The **w** function modifier and the –**C** option of *tar*, for pausing or changing directories between the files listed on the command line.

- Writing and reading *tttarfiles* (that is, archives produced with the **L** function modifier) directly to and from magnetic tape.

pathname
A pathname of a regular file or directory to be archived (when the **c** function letter is used), extracted (**x**) or listed (**t**). When *pathname* is the pathname of a directory, the action applies to all of the files and (recursively) subdirectories of that directory. When the **f** letter is used in the *key* operand, the initial *pathname* operand is interpreted as an archive name, as described previously.

tarfile
A pathname of a regular file to be read or written as an archive of files.

ttarfile
A pathname of a regular file to be read or written as an archive of ToolTalk objects.

STDIN

When the **f** modifier is used with the **t** or **x** function letter and the pathname is –, the standard input is an archive file formatted as described in **EXTENDED DESCRIPTION**. Otherwise, the standard input is not used.

INPUT FILES

The files identified by the *pathname* operands are regular files or directories. The file identified by the *tarfile* operand is a regular file formatted as described in *tar* in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**. The file identified by the *tttarfile* operand is a regular file formatted as described in **EXTENDED DESCRIPTION**.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tttar*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility behaves as if none of the variables had been defined.

LC_ALL If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_MESSAGES Determine the locale that is used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

NLSPATH Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used with date and time strings.

RESOURCES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the –**h** option is used, *tttar* writes to standard output a help message in an unspecified format.

When the –**v** option is used, *tttar* writes to standard output a version number in an unspecified format.

When the **f** modifier is used with the **c** function letter and the pathname is –, the standard output is an archive file formatted as described in **EXTENDED DESCRIPTION**.

Otherwise, the standard output is not used.

STDERR

The standard error is used for diagnostic messages and the file name output described under the **v** modifier (when the **t** function letter is not used).

OUTPUT FILES

Output files are created, as specified by the archive, when the **x** function letter is used.

EXTENDED DESCRIPTION

The archive file produced and read by *tttar* is formatted as described in *tar*, with the addition of one extra file named **tttarfile**. (If one of the user files being archived is also named **tttarfile**, the results are unspecified.) The **tttarfile** contains all the ToolTalk *spec* information for the ToolTalk objects in the other files in the archive. The contents of **tttarfile** are written according to the referenced XDR specification (RFC 1014). The only XDR data types used are:

int A four-octet signed integer, most significant octet first

string A four-octet unsigned integer length, most significant octet first, followed by the characters of the string, followed by sufficient (0 to 3) residual zero octets to make the total number of octets a multiple of four.

The **tttarfile** starts with two integers. The first is always 1, to mark this as the header record. The second is always 1, indicating this is version 1 of the *tttarfile* format.

The end of the **tttarfile** is a integer 3, marking the end-of-file record.

In between, there is one logical record for each spec. Each logical record starts with an integer 2, marking it as a spec record. Other integer values are reserved for assignment to future data types.

After the record identifier, the spec record contains, in sequence:

1. A string giving the Tooltalk object identifier (*objid*) of the object represented by the spec
2. A string giving the name of the file (as found in the archive table of contents) that contains the contents of the ToolTalk object represented by the spec
3. A string giving the ToolTalk object type identifier (*otid*) of the ToolTalk object represented by the spec
4. An integer giving the number of properties for this object

The properties of the object immediately follow the number of properties. Each property consists of:

1. A string giving the name of the property
2. An integer, which is always zero (for historical compatibility)
3. An integer giving the number of values for this property
4. A string for each value

After the values, the next property is found, until all properties for the object have been accounted for; then the next spec is found, until all specs for objects associated with files in the archive are accounted for.

EXIT STATUS

The following exit values are returned:

0 All files and ToolTalk objects were moved successfully.

>0 An error occurred or the invoked *tar* command exited with a non-zero value.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

SEE ALSO

tar in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**; *ttcp*, *ttsession*.

CHANGE HISTORY

First released in Issue 1.

6.5 Data Formats

This section defines the data formats of static message patterns, iX "data formats" "static message patterns" iX "static message patterns" used in ptype and otype files.

The static messaging method allows an application to specify the message pattern information if it wants to receive a defined set of messages. To use the static method, the application must define its process types and object types and compile them with the ToolTalk type compiler, *tt_type_comp*. When the application declares its process type, the ToolTalk service creates message patterns based on that type. These static message patterns remain in effect until the application closes communication with the ToolTalk service.

6.5.1 Defining Process Types

An application can be considered a potential message receiver even when no process is running the application. To do this, the application developer must provide message patterns and instructions on how to start the application in a process type (ptype) file. These instructions tell the ToolTalk service to perform one of the following actions when a message is available for an application but the application is not running:

- Start the application and deliver the message
- Queue the message until the application is running
- Discard the message

To make the information available to the ToolTalk service, the ptype file must be compiled with the ToolTalk type compiler, *tt_type_comp*, at application installation time.

When an application registers a ptype with the ToolTalk service, the message patterns listed in it are automatically registered, too.

Ptypes provide application information that the ToolTalk service can use when the application is not running. This information is used to start your process if necessary to receive a message or queue messages until the process starts.

A ptype begins with a process-type identifier (ptid). Following the ptid are:

1. An optional start command string, which the ToolTalk service will execute, if necessary, to start a process running the program.
2. Signatures, which describe the TT_PROCEDURE-addressed messages that the program wants to receive. Messages to be observed are described separately from messages to be handled.

Signatures

Signatures describe the messages that the program wants to receive. A signature is divided by an arrow (the two characters =>) into two parts. The first part of a signature specifies matching attribute values. The more attribute values specified in a signature, the fewer messages the signature will match. The second part of a signature specifies receiver values that the ToolTalk service will copy into messages that match the first part of the signature.

A ptype signature can contain values for disposition and operation numbers (opnums). The ToolTalk service uses the disposition value (**start**, **queue** or the default **discard**) to determine what to do with a message that matches the signature when no process is running the program. The opnum value is provided as a convenience to message receivers. When two signatures have the same operation name but different arguments, different opnums makes incoming messages easy to identify.

The following is the syntax for a ptype file.

```
ptype         ::=   'ptype' ptid '{'
                    property*
                    ['observe:' psignature*]
                    ['handle:' psignature* ]
                    '}' [';']
property      ::=   property_id value ';'
property_id   ::=   'start'
value         ::=   string
ptid          ::=   identifier
psignature    ::=   [scope] op args [contextdcl]
                    ['=>'
                    ['start']['queue']
                    ['opnum='number]]
                    ';'
scope         ::=   'file'
                |   'session'
                |   'file_in_session'
args          ::=   '(' argspec {, argspec}* ')'
                |   '(void)'
                |   '()'
contextdcl    ::=   'context' '(' identifier {, identifier}* ')' ';'
argspec       ::=   mode type name
mode          ::=   'in' | 'out' | 'inout'
type          ::=   identifier
name          ::=   identifier
```

Property_id Information

ptid The process type identifier (ptid) identifies the process type. A ptid must be unique for every implementation. It is recommended that the name selected include the trademarked name of the application product or company as a prefix. The ptid cannot exceed 32 bytes and must not be one of the reserved identifiers: **ptype**, **otype**, **start**, **opnum**, **queue**, **file**, **session**, **observe** or **handle**.

start The start string for the process. If the ToolTalk service needs to start a process, it executes this command using the shell described in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**.

Before executing the command, the ToolTalk service defines *TT_FILE* as an environment variable with the value of the file attribute of the message that started the application. This command runs in the environment of *ttsession*, not in the environment of the sender of the message that started the application, so any context information must be carried by message arguments or contexts.

Psignature Matching Information

scope This pattern attribute is matched against the scope attribute in messages.

op Operation name. This name is matched against the op attribute in messages.

Note: If the application specifies message signatures in both its ptype and otypes, it must use unique operation names in each. For example, it cannot specify a display operation in both its ptype and otype.

args Arguments for the operation. If the argument list is **(void)**, the signature matches only messages with no arguments. If the argument list is empty (that is, ''()''), the signature matches without regard to the arguments.

contextdcl
The context name. When a pattern with this named context is generated from the signature, it contains an empty value list.

Psignature Actions Information

start If the psignature matches a message and no running process of this ptype has a pattern that matches the message, the ToolTalk service starts a process of this ptype.

queue If the psignature matches a message and no running process of this ptype has a pattern that matches the message, the ToolTalk service queues the message until a process of this ptype registers a pattern that matches it.

opnum The application should fill in the message's opnum attribute with the specified number to enable it to identify the signature that matched the message. When the message matches the signature, the the ToolTalk service fills the opnum from the signature into the message. The application can then retrieve the opnum with the *tt_message_opnum*( ) call. By giving each signature a unique opnum, the application can determine which signature matched the message. It can attach a callback routine to the opnum with the *tt_ptype_opnum_callback_add* ( ) call. When the message is matched, the ToolTalk service will check for any callbacks attached to the opnum and, if any are found, run them.

6.5.2 Defining Object Types

When a message is addressed to a specific object or a type of object, the ToolTalk service must be able to determine to which application the message is to be delivered. Applications provide this information in an object type (otype). An otype names the ptype of the application that manages the object and describes message patterns that pertain to the object. These message patterns also contain instructions that tell the ToolTalk service what to do if a message is available but the application is not running. In this case, the ToolTalk service performs one of the following instructions:

- Start the application and deliver the message
- Queue the message until the application is running
- Discard the message

To make the information available to the ToolTalk service, the otype file must be compiled with the ToolTalk type compiler *tt_type_comp* at application installation time. When an application that manages objects registers with the ToolTalk service, it declares its ptype. When a ptype is registered, the ToolTalk service checks for otypes that mention the ptype and registers the patterns found in these otypes. The otype for the application provides addressing information that the ToolTalk service uses when delivering object-oriented messages. The number of otypes, and what they represent, depends on the nature of the application. For example, a word

processing application might have otypes for characters, words, paragraphs and documents; a diagram editing application might have otypes for nodes, arcs, annotation boxes and diagrams.

An otype begins with an object-type identifier (otid). Following the otid are:

1. An optional start command string, which the ToolTalk service will execute, if necessary, to start a process running the program.
2. Signatures, which define the messages that can be addressed to objects of the type (that is, the operations that can be invoked on objects of the type).

Signatures

Signatures define the messages that can be addressed to objects of the type. A signature is divided by an arrow (the two characters =>) into two parts. The first part of a signature define matching criteria for incoming messages. The second part of a signature defines receiver values that the ToolTalk service adds to each message that matches the first part of the signature. These values specify the ptid of the program that implements the operation and the message's scope and disposition.

Creating Otype Files

The following is the syntax for an otype file:

```
otype         ::=   obj_header   '{' objbody* '}' [';']
obj_header    ::=   'otype' otid [':' otid+]
objbody       ::=   'observe:' osignature*
                |   'handle:' osignature*

osignature    ::=   op args [contextdcl] [rhs][inherit] ';'
rhs           ::=   ['=>' ptid [scope]]
                    ['start']['queue']
                    ['opnum='number]
inherit       ::=   'from' otid
args          ::=   '(' argspec {, argspec}* ')'
                |   '(void)'
                |   '()'
contextdcl    ::=   'context' '(' identifier {, identifier}* ')' ';'
argspec       ::=   mode type name
mode          ::=   'in' | 'out' | 'inout'
type          ::=   identifier
name          ::=   identifier
otid          ::=   identifier
ptid          ::=   identifier
```

Obj_Header Information

otid The object type identifier (otid) identifies the object type. A otid must be unique for every implementation. It is recommended that the identifier begin with the ptid of the tool that implements the otype. The otid is limited to 64 bytes and must not be one of the reserved identifiers: **ptype**, **otype**, **start**, **opnum**, **start**, **queue**, **file**, **session**, **observe** or **handle**.

Osignature Information

The object body portion of the otype definition is a list of osignatures for messages about the object that the application wants to observe and handle.

op Operation name. This name is matched against the op attribute in messages.

args Arguments for the operation. If the argument list is **(void)**, the signature matches only messages with no arguments. If the argument list is empty (just ''()''), the signature matches messages without regard to the arguments.

contextdcl
Context name. When a pattern with this named context is generated from the signature, it contains an empty value list.

ptid The process type identifier for the application that manages this type of object.

opnum The application should fill in the message's opnum attribute with the specified number to enable it to identify the signature that matched the message. When the message matches the signature, the the ToolTalk service fills the opnum from the signature into the message. The application can then retrieve the opnum with the *tt_message_opnum*( ) call. By giving each signature a unique opnum, the application can determine which signature matched the message. It can attach a callback routine to the opnum with the *tt_ptype_opnum_callback_add* ( ) call. When the message is matched, the ToolTalk service will check for any callbacks attached to the opnum and, if any are found, run them.

inherit Otypes form an inheritance hierarchy in which operations can be inherited from base types. The ToolTalk service requires the otype definer to name explicitly all inherited operations and the otype from which to inherit. This explicit naming prevents later changes (such as adding a new level to the hierarchy, or adding new operations to base types) from unexpectedly affecting the behaviour of an otype.

scope This pattern attribute is matched against the scope attribute in messages. It appears on the rightmost side of the arrow and is filled in by the ToolTalk service during message dispatch. This means the definer of the otype can specify the attributes instead of requiring the message sender to know how the message should be delivered.

Osignature Actions Information

start If the osignature matches a message and no running process of this otype has a pattern that matches the message, the ToolTalk service will start a process of this otype.

queue If the osignature matches a message and no running process of this otype has a pattern that matches the message, the ToolTalk service will queue the message until a process of this otype registers a pattern that matches it.

6.6 Protocol Message Sets

This section describes standard ToolTalk messages. Many of the XCDE services can be controlled or accessed by sending them ToolTalk messages; those services that do support ToolTalk interaction list the messages in a section named ''Messages'' in the appropriate chapters of the **XCSA** specification.

Each message is described on a separate reference page, similar to the format used for a C-language function. The **SYNOPSIS** section is a representation of the message in a syntax similar to that understood by the ToolTalk type compiler, *tt_type_comp*( ). The synopsis format is:

```
[file] opName(requiredArgs, [optionalArgs]);
```

The components of the synopsis are as follows:

file If the synopsis begins with [*file*], this is an indication that the file attribute of the message can or should be set. ToolTalk allows each message to refer to a file, and has a mechanism (called file-scoping) for delivering messages to clients who are ''interested'' in the named file. See the *tt_message_file_set*( ) function.

opName
The name of the operation or event.

requiredArgs, optionalArgs
In the synopsis, arguments are expressed as:

```
mode vtype argumentName
```

The *mode* part is one of **in**, **out** or **inout**, indicating the direction(s) in which the data of that argument flow.

The *vtype* and *argumentName* parts describe a particular argument. The *vtype* is a programmer-defined string that describes what kind of data a message argument contains. ToolTalk uses vtypes for the sole purpose of matching sent message instances with registered message patterns. Every vtype should by convention map to a single, well-known data type. The data type of a ToolTalk argument is either integer, string or bytes. The data type of a message or pattern argument is determined by the ToolTalk API function used to set its value. The *argumentName* is merely a comment hinting to human readers at the semantics of the argument, much like a parameter name in an ISO C function prototype.

The *requiredArgs* shown without [ ] brackets are required to form a valid message. The *optionalArgs* shown enclosed in [ ] brackets are optional. The extra arguments that may be included in a message. Any optional arguments in a message must be in the specified order, and must follow the required arguments.

The **ERRORS** section describes integer status codes that can be extracted from a reply via *tt_message_status*( ). This status defaults to zero (TT_OK), or can be set by the handler via *tt_message_status_set*( ). In extraordinary circumstances such as no matching handler, ToolTalk itself sets the message status to a **Tt_status** code.

6.6.1 Desktop Message Set

The *Desktop* message policies apply to any tool in an XPG4 or X Window System environment. In addition to standard messages for these environments, the *Desktop* policies define data types and error codes that apply to all of the ToolTalk message policies.

The following types and argument names are used in message **SYNOPSIS** descriptions:

boolean
: A vtype for logical values. The underlying data type of boolean is integer; that is, arguments of this vtype should be manipulated with *tt_*_arg_ival*[*_set*]( ) and *tt_*_iarg_add( )* functions. Zero means false; non-zero means true.

string
: A vtype for character strings. Arguments of this vtype should be manipulated with *tt_*_arg_val*[*_set*]( ) and *tt_*_arg_add*( ) functions.

messageID
: A vtype for uniquely identifying messages. The underlying data type of **messageID** is **string**. The **messageID** of a **Tt_message** is returned by *tt_message_id*( ).

width
height
xOffset
yOffset
: Vtypes for integer geometry values, in pixels.

type
: Any of the vtypes that are the name of the kind of objects in a particular system of persistent objects. For example, the vtype for the kind of objects in file systems is **File**. The vtype for ToolTalk objects is **ToolTalk_Object**.

NAME

Get_Environment request — get a tool's environment

SYNOPSIS

```
Get_Environment(in string variable,
                out string value
                [...]);
```

DESCRIPTION

The *Get_Environment* request reports the value of the indicated environment variable(s).

The *variable* argument is the name of the environment variable to get.

The *value* argument is the value of the environment variable. If no value (in other words, **(char *)0**) is returned for this argument, then the variable was not present in the handler's environment. This condition is not an error. If an empty string (in other words, "") is returned for this argument, then the variable was present in the handler's environment, but had a null value.

APPLICATION USAGE

The *ttdt_session_join*( ), function can be used to register for, and transparently process, the *Get_Environment* request.

SEE ALSO

ttdt_session_join( ); *Set_Environment* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Get_Geometry request — get a tool's on-screen geometry

SYNOPSIS

```
Get_Geometry(out width w,
             out height h,
             out xOffset x,
             out yOffset y
             [in messageID commission]);
```

DESCRIPTION

The *Get_Geometry* request reports the on-screen geometry of the optionally specified window, or of the window primarily associated with the recipient procid (if no window is specified).

The *w*, *h*, *x* and *y* arguments are integer geometry values, in pixels, representing width, height, x-coordinate and y-coordinate, respectively. Negative offset values are interpreted according to the X/Open CAE Specification, **Window Management: Xlib — C Language Binding**.

The *commission* argument is the ID of the ongoing request, if any, that resulted in the creation of the window in question.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Get_Geometry* request. Also, *Get_Geometry* can be sent by *ttdt_sender_imprint_on*( ).

SEE ALSO

ttdt_message_accept( ), *ttdt_sender_imprint_on*( ), *ttdt_session_join*( ); *Set_Geometry* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Get_Iconified request — get a tool's iconic state

SYNOPSIS

```
Get_Iconified(out boolean iconic
              [in messageID commission]);
```

DESCRIPTION

The *Get_Iconified* request reports the iconic state of the optionally specified window, or of the window primarily associated with the handling procid (if no window is specified).

The *iconic* argument is a Boolean value indicating whether the specified window is (to be) iconified.

The *commission* argument is the ID of the ongoing request, if any, that resulted in the creation of the window(s) in question.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Get_Iconified* request.

SEE ALSO

ttdt_message_accept( ), *ttdt_session_join*( ); *Set_Iconified* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Get_Locale request — get a tool's locale

SYNOPSIS

```
Get_Locale(in string category,
           out string locale
           [...]);
```

DESCRIPTION

The *Get_Locale* request reports the XPG4 locale of the indicated locale categories.

The *category* argument is the locale category to get. A locale category is a group of data types whose output formatting varies according to locale in a similar way. ISO C and X/Open locale categories are:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
```

The *locale* argument is the name of the current locale of the indicated category. The value of *locale* is implementation-defined.

ERRORS

The ToolTalk service may return the following error in processing the *Get_Locale* request:

TT_DESKTOP_EINVAL
: The *locale* argument is not valid on the handler's host.

APPLICATION USAGE

The *ttdt_session_join*( ), function can be used to register for, and transparently process, the *Get_Locale* request.

Also, *Get_Locale* can be sent by *ttdt_sender_imprint_on*( ), with the reply being handled transparently.

SEE ALSO

setlocale( ) in the , *ttdt_sender_imprint_on*( ), *ttdt_session_join*( ); *Set_Locale* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Get_Mapped request — get whether a tool is mapped to the screen

SYNOPSIS

```
Get_Mapped(out boolean mapped
           [in messageID commission]);
```

DESCRIPTION

The *Get_Mapped* request reports the mapped state of the optionally specified window, or of the window primarily associated with the handling procid (if no window is specified).

The *mapped* argument is a Boolean value indicating whether the specified window is (to be) mapped to the screen.

The *commission* argument is the ID of the ongoing request, if any, that resulted in the creation of the window in question.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Get_Mapped* request.

SEE ALSO

ttdt_message_accept( ), *ttdt_session_join*( ); *Set_Mapped* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Get_Modified request — ask whether an entity has been modified

SYNOPSIS

```
[file] Get_Modified(in type ID,
                    out boolean modified);
```

DESCRIPTION

The *Get_Modified* request asks whether any tool has modified a volatile, non-shared (for example, in-memory) representation of the persistent state of an entity (such as a file) with the intention of eventually making that representation persistent.

Thus, a tool should register a dynamic pattern for this request when it has modified an entity of possible shared interest.

The *ID* argument is the identity of the persistent entity being asked about. When its *type* is **File**, then *ID* is unset (in other words, has a value of **(char *)0**), and it refers to the file or directory named in the message's file attribute.

The *modified* argument argument is a Boolean value indicating whether a volatile, non-shared (for example, in-memory) representation of the entity has been modified with the intention of eventually making that representation persistent.

ERRORS

The ToolTalk service may return one of the following errors in processing the *Get_Modified* request:

TT_ERR_NO_MATCH
: Since no handler could be found, the entity in question can be assumed not to be modified.

APPLICATION USAGE

The *ttdt_file_join*( ), function can be used to register for, and transparently process, the *Get_Modified* request.

The *Get_Modified* request can be sent with *ttdt_file_request*( ); *ttdt_Get_Modified*( ), can send the *Get_Modified* request and block on the reply.

SEE ALSO

ttdt_file_join( ), *ttdt_file_request*( ), *ttdt_file_request*( ), *ttdt_Get_Modified*( ); *Set_Modified* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Get_Situation request — get a tool's current working directory

SYNOPSIS

```
Get_Situation(out string path);
```

DESCRIPTION

The *Get_Situation* request reports the current working directory.

The *path* argument is the pathname of the working directory that the recipient is using.

APPLICATION USAGE

The *ttdt_session_join*( ), function can be used to register for, and transparently process, the *Get_Situation* request.

SEE ALSO

ttdt_session_join( ); *Set_Situation* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Get_Status request — retrieve a tool's current status

SYNOPSIS

```
Get_Status(out string status,
           out string vendor,
           out string toolName,
           out string toolVersion
           [in messageID operation2Query]);
```

DESCRIPTION

The *Get_Status* request retrieves the current status of a tool (or, optionally, of a specific operation being performed by a tool).

The *status* argument is the status retrieved.

The *vendor* argument is the name of the vendor of the handling tool.

The *toolName* argument is the name of the handling tool.

The *toolVersion* argument is the version of the handling tool.

OPTIONAL

The *operation2Query* argument is the ID of the request that initiated the operation the status of which is being requested.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used to register for, and help process, the *Get_Status* request.

EXAMPLES

After sending a TT_REQUEST and storing its handle in **Tt_message** *request_I_sent*, if the handler identifies itself with a *Status* notice saved in **Tt_message** *status_msg_from_handler*, then the status of *request_I_sent* can be queried as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
             tt_message_sender(status_msg_from_handler),
             TTDT_GET_STATUS, my_callback);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_IN, Tttk_string,
                    tt_message_id(request_I_sent));
tt_message_send(msg);
```

SEE ALSO

tt_message_arg_add( ), *tt_message_id*( ), *tt_message_send*( ), *ttdt_message_accept*( ), *tt_message_sender*( ), *ttdt_session_join*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

Get_Sysinfo request — get information about a tool's host

SYNOPSIS

```
Get_Sysinfo(out string sysname,
            out string nodename,
            out string release,
            out string version,
            out string machine);
```

DESCRIPTION

The *Get_Sysinfo* request gets information about the handler's host.

The *sysname* argument is the name of the host's operating system.

The *nodename* argument is the name of the host.

The *release* and *version* arguments are implementation-specific information about the host's operating system.

The *machine* argument is an implementation-specific name that identifies the hardware on which the operating system is running.

APPLICATION USAGE

The *ttdt_session_join*( ), function can be used to register for, and transparently process, the *Get_Sysinfo* request.

EXAMPLES

The *Get_Sysinfo* message can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_GET_SYSINFO,
                    my_callback);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_send(msg);
```

SEE ALSO

uname( ) in the X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**, *tt_message_arg_add*( ), *tt_message_send*( ), *ttdt_session_join*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

Get_XInfo request — get a tool's X11 attributes

SYNOPSIS

```
Get_XInfo(out string display,
          out string visual,
          out integer depth
          [in messageID commission]);
```

DESCRIPTION

The *Get_XInfo* request reports the X11 attributes of the optionally specified window, or of the window primarily associated with the recipient procid (if no window is specified).

The *display* argument is an X11 display.

The *visual* argument is an X11 visual class (which determines how a pixel will be displayed as a colour). Valid values are:

```
DirectColor   PseudoColor   StaticGray
GrayScale     StaticColor   TrueColor
```

The *depth* argument is the number of bits in a pixel.

The *commission* argument is the ID of the ongoing request with respect to which X11 attributes are being set or reported.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Get_XInfo* request. Also, *Get_XInfo* can be sent by *ttdt_sender_imprint_on*( ).

Since the handler may be running on a different host, it is almost always better to return a *display* value of *hostname*:*n*[.*n*] instead of :*n*[.*n*].)

The *commission* argument is useful to the extent that the handler employs different attributes for the different operations it may be carrying out.

EXAMPLES

The *Get_XInfo* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_GET_XINFO,
                    my_callback);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_iarg_add(msg, TT_OUT, Tttk_integer, 0);
tt_message_send(msg);
```

SEE ALSO

tt_message_iarg_add( ), *tt_message_send*( ), *ttdt_message_accept*( ), *ttdt_sender_imprint_on*( ), *ttdt_session_join*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

Lower request — lower a tool's window(s) to the back

SYNOPSIS

```
Lower([in messageID commission]);
```

DESCRIPTION

The *Lower* request lowers the window(s) associated with the handling procid. If any optional arguments are present, then it lowers only the indicated window(s).

The *commission* argument is the ID of the message, if any, that resulted in the creation of the window(s) that should be lowered.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Lower* request.

SEE ALSO

ttdt_message_accept( ), *ttdt_session_join*( ); *Raise* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Modified notice — an entity has been modified

SYNOPSIS

```
[file] Modified(in type ID);
```

DESCRIPTION

The *Modified* notice is sent whenever a tool first modifies a volatile, non-shared (for example, in-memory) representation of the persistent state of an entity (such as a file), with the intention of eventually making that representation persistent.

The *ID* argument is the identity of the modified entity. When its *type* is **File**, then the *ID* argument is unset (in other words, has a value of **(char *)0**), and it refers to the file or directory named in the message's file attribute.

APPLICATION USAGE

The *ttdt_file_join*( ), function can be used to register for, and help process, the *Modified* request.

The *Modified* request can be sent with *ttdt_file_event*( ).

SEE ALSO

ttdt_file_event( ). *ttdt_file_join*( ); *Reverted* notice.

CHANGE HISTORY

First released in Issue 1.

NAME

Pause request — pause a tool, operation or data performance

SYNOPSIS

```
Pause([in messageID operation]);
```

DESCRIPTION

The *Pause* request pauses the specified tool, operation or data performance.

If the optional *operation* argument is included, the handler should pause the operation that was invoked by the specified request.

The *operation* argument is the request that should be paused. For a request to be eligible for pausing, the handler must have sent a *Status* notice back to the requester (thus identifying itself to the requester).

ERRORS

The ToolTalk service may return the following error in processing the *Pause* request:

TT_DESKTOP_ENOMSG
: The *operation* argument does not refer to any message currently known by the handler.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used to register for, and help process, the *Pause* request.

EXAMPLES

The *Pause* message can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                     the_recipient_procid, TTDT_PAUSE,
                     my_callback);
tt_message_send(msg);
```

SEE ALSO

tt_message_send( ), *ttdt_message_accept*( ), *ttdt_session_join*( ); *Resume* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Quit request — terminate an operation or an entire tool

SYNOPSIS

```
Quit(in boolean silent,
     in boolean force
     [in messageID operation2Quit]);
```

DESCRIPTION

The *Quit* request terminates an operation or an entire tool. Without the optional *operation2Quit* argument, this request asks the handling procid to quit. If the request succeeds, one or more ToolTalk procids should call *tt_close*( ), and zero or more processes should exit.

With the optional *operation2Quit* argument, this request asks the handler to terminate the indicated request. (Whether the terminated request must therefore be failed depends on its semantics. Often, termination can be considered to mean that the requested operation has been carried out to the requester's satisfaction.)

The *Quit* request should be failed (and the status code set appropriately) when the termination is not performed—for example, because the *silent* argument was false and the user canceled the quit.

The *silent* argument affects user notification of termination. If *silent* is True, the handler is not allowed to block on user input before terminating itself (or the indicated operation). If it is False, however, the handler may seek such input.

The *force* argument is a Boolean value indicating whether the handler should terminate itself (or the indicated operation) even if circumstances are such that the tool ordinarily would not perform the termination.

For example, a tool might have a policy of not quitting with unsaved changes unless the user has been asked whether the changes should be saved. When *force* is true, such a tool should terminate even when doing so would lose changes that the user has not been asked by the tool about saving.

The *operation2Quit* argument is the request that should be terminated. For a request to be terminable, the handler must have sent a *Status* notice back to the requester (thus identifying itself to the requester).

ERRORS

The ToolTalk service may return one of the following errors in processing the *Quit* request:

TT_DESKTOP_ECANCELED
: The user overrode the *Quit* request.

TT_DESKTOP_ENOMSG
: The *operation2Quit* argument does not refer to any message currently known by the handler.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used to register for, and help process, the *Quit* request.

In the successful case, ''zero or more'' procids are cited because a single process can instantiate multiple independent procids, and a single procid can conceivably be implemented by a set of cooperating processes.

EXAMPLES

The *Quit* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_QUIT,
                    my_callback);
tt_message_iarg_add(msg, TT_IN, Tttk_boolean, 0);
tt_message_iarg_add(msg, TT_IN, Tttk_boolean, 0);
tt_message_send(msg);
```

SEE ALSO

tt_close( ), *tt_message_iarg_add*( ), *tt_message_send*( ), *ttdt_message_accept*( ), *ttdt_session_join*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

Raise request — raise a tool's window(s) to the front

SYNOPSIS

```
Raise([in messageID commission]);
```

DESCRIPTION

The *Raise* request raises the window(s) associated with the handling procid. If any optional arguments are present, then it raises only the indicated window(s).

The *commission* argument is the ID of the message, if any, that resulted in the creation of the window(s) that should be raised.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Raise* request.

EXAMPLES

The *Raise* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_RAISE,
                    my_callback);
tt_message_send(msg);
```

SEE ALSO

tt_message_send( ), *ttdt_message_accept*( ), *ttdt_session_join*( ); *Lower* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Resume request — resume a tool, operation or data performance

SYNOPSIS

```
Resume([in messageID operation]);
```

DESCRIPTION

The *Resume* request resumes the specified tool, operation or data performance.

If the optional *operation* argument is included, the handler should resume the operation that was invoked by the specified request.

The *operation* argument is the request that should be resumed.

ERRORS

The ToolTalk service may return the following error in processing the *Resume* request:

TT_DESKTOP_ENOMSG
: The *operation* argument does not refer to any message currently known by the handler.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used to register for, and help process, the *Resume* request.

SEE ALSO

ttdt_message_accept( ), *ttdt_session_join*( ); *Pause* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Revert notice — discard any modifications to an entity

SYNOPSIS

```
[file] Revert(in type ID);
```

DESCRIPTION

The *Revert* notice asks that any pending, unsaved modifications to a persistent entity (such as a file) be discarded.

The *ID* argument is the identity of the entity to revert. When its *type* is **File**, then the *ID* argument is unset (in other words, has a value of **(char *)0**), and it refers to the file or directory named in the message's file attribute.

ERRORS

The ToolTalk service may return one of the following errors in processing the *Revert* notice:

TT_DESKTOP_UNMODIFIED
: The entity had no pending, unsaved modifications.

TT_DESKTOP_ENOENT
: The file to save/revert does not exist.

APPLICATION USAGE

The *ttdt_file_join*( ), function can be used to register for, and help process, the *Revert* request.

The *Revert* request can be sent with *ttdt_file_request*( ). Also, *ttdt_Revert*( ), can send the relevant message and block on the reply.

SEE ALSO

ttdt_Revert( ), *ttdt_file_join*( ), *ttdt_file_request*( ); *Save* notice.

CHANGE HISTORY

First released in Issue 1.

NAME

Reverted notice — an entity has been reverted

SYNOPSIS

```
[file] Reverted(in type ID);
```

DESCRIPTION

The *Reverted* notice is sent when all the modifications (see the *Modified* notice) to an entity have been discarded.

The *ID* argument is the identity of the modified or reverted entity. When its *type* is **File**, then the *ID* argument is unset (in other words, has a value of **(char *)0**), and it refers to the file or directory named in the message's file attribute.

APPLICATION USAGE

The *ttdt_file_join*( ), function can be used to register for, and help process, the *Reverted* request.

The *Reverted* request can be sent with *ttdt_file_event*( ).

SEE ALSO

ttdt_file_event( ), *ttdt_file_join*( ); *Saved* notice.

CHANGE HISTORY

First released in Issue 1.

NAME

Save notice — save any modifications to an entity

SYNOPSIS

[*file***]** Save(in *type ID*);

DESCRIPTION

The *Save* notice asks that any pending, unsaved modifications to a persistent entity (such as a file) be saved.

The *ID* argument is the identity of the entity to save. When its *type* is **File**, then the *ID* argument is unset (in other words, has a value of **(char** ∗**)0**), and it refers to the file or directory named in the message's file attribute.

ERRORS

The ToolTalk service may return one of the following errors in processing the *Save* notice:

TT_DESKTOP_UNMODIFIED
: The entity had no pending, unsaved modifications.

TT_DESKTOP_ENOENT
: The file to save/revert does not exist.

APPLICATION USAGE

The *ttdt_file_join*( ), function can be used to register for, and help process, the *Save* request.

The *Save* request can be sent with *ttdt_file_request*( ). Also, *ttdt_Save*( ), can send the relevant message and block on the reply.

SEE ALSO

ttdt_Save( ), *ttdt_file_join*( ), *ttdt_file_request*( ); *Revert* notice.

CHANGE HISTORY

First released in Issue 1.

NAME

Saved notice — an entity has been saved to persistent storage

SYNOPSIS

```
[file] Saved(in type ID);
```

DESCRIPTION

The *Saved* notice announces that the persistent storage for an entity (such as a file) has been updated.

The *ID* argument is the identity of the saved entity. When its *type* is **File**, then the *ID* argument is unset (in other words, has a value of **(char *)0**), and it refers to the file or directory named in the message's file attribute.

APPLICATION USAGE

The *ttdt_file_join*( ), function can be used to register for, and help process, the *Saved* request.

The *Saved* request can be sent with *ttdt_file_event*( ).

SEE ALSO

ttdt_file_event( ), *ttdt_file_join*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

Set_Environment request — set a tool's environment

SYNOPSIS

```
Set_Environment(in string variable,
                in string value
                [...]);
```

DESCRIPTION

The *Set_Environment* request replaces the value of the indicated environment variable(s).

The *variable* argument is the name of the environment variable to set.

The *value* argument is the value of the environment variable. If this argument is unset (in other words, has a value of **(char *)0**), then the variable should be removed from the environment. It is not an error for the variable not to have existed in the first place.

APPLICATION USAGE

The *ttdt_session_join*( ), function can be used to register for, and transparently process, the *Set_Environment* request.

EXAMPLES

The *Set_Environment* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_SET_ENVIRONMENT,
                    my_callback);
tt_message_arg_add(msg, TT_IN, Tttk_string, "PATH");
tt_message_arg_add(msg, TT_IN, Tttk_string, "/bin:/usr/ucb");
tt_message_send(msg);
```

SEE ALSO

tt_message_arg_add( ), *tt_message_send*( ), *ttdt_session_join*( ); *Get_Environment* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Set_Geometry request — set a tool's on-screen geometry

SYNOPSIS

```
Set_Geometry(inout width w,
             inout height h,
             inout xOffset x,
             inout yOffset y
             [in messageID commission]);
```

DESCRIPTION

The *Set_Geometry* request sets the on-screen geometry of the optionally specified window, or of the window primarily associated with the recipient procid (if no window is specified).

The *w*, *h*, *x* and *y* arguments are integer geometry values, in pixels, representing width, height, x-coordinate and y-coordinate, respectively. Negative offset values are interpreted according to the X/Open CAE Specification, **Window Management: Xlib — C Language Binding**. If any of these arguments are unset, that part of the geometry need not be changed. The return values are the actual new values, in case they differ from the requested new values.

The *commission* argument is the ID of the ongoing request, if any, that resulted in the creation of the window in question.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Set_Geometry* request.

EXAMPLES

The *Set_Geometry* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_SET_GEOMETRY,
                    my_callback);
tt_message_iarg_add(msg, TT_INOUT, Tttk_width, 500);
tt_message_iarg_add(msg, TT_INOUT, Tttk_height, 500);
tt_message_arg_add(msg, TT_INOUT, Tttk_xoffset, 0); /* no value */
tt_message_arg_add(msg, TT_INOUT, Tttk_yoffset, 0); /* no value */
tt_message_send(msg);
```

SEE ALSO

tt_message_arg_add( ), *tt_message_iarg_add*( ), *tt_message_send*( ), *ttdt_message_accept*( ), *ttdt_session_join*( ); *Get_Geometry* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Set_Iconified request — set a tool's iconic state

SYNOPSIS

```
Set_Iconified(in boolean iconic
              [in messageID commission]);
```

DESCRIPTION

The *Set_Iconified* request sets the iconic state of the optionally specified window, or of the window primarily associated with the handling procid (if no window is specified).

The *iconic* argument is a Boolean value indicating whether the specified window is (to be) iconified.

The *commission* argument is the ID of the ongoing request, if any, that resulted in the creation of the window(s) in question.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Set_Iconified* request.

If the handler does not map window-system windows one-to-one to commissions or procids, then it may interpret ''iconic state'' liberally. For example, consider a *Display* request on an **ISO_Latin_1** file, handled by a *gnuemacs* instance that then devotes an *emacs* ''window'' to the file. ''Windows'' in *gnuemacs* are not separate X11 windows, and are not separately iconifiable. However, a *Set_Iconified* request issued with respect to the ongoing *Display* request could be liberally interpreted by *gnuemacs* to mean it should minimise the screen real estate devoted to the operation, perhaps by ''burying'' the buffer or dividing its window's real estate among neighbouring windows. And, if the *Display* request happens to be the only thing *emacs* is working on at the moment, it could instead take a literal interpretation, and actually iconify itself.

EXAMPLES

The *Set_Iconified* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_SET_ICONIFIED,
                    my_callback);
tt_message_iarg_add(msg, TT_IN, Tttk_boolean, 1);
tt_message_send(msg);
```

SEE ALSO

tt_message_iarg_add( ), *tt_message_send*( ), *ttdt_message_accept*( ), *ttdt_session_join*( ); *Get_Iconified* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Set_Locale request — set a tool's locale

SYNOPSIS

```
Set_Locale(in string category,
           in string locale
           [...]);
```

DESCRIPTION

The *Set_Locale* request reports the XPG4 locale of the indicated locale categories.

The *category* argument is the locale category to set. A locale category is a group of data types whose output formatting varies according to locale in a similar way. ISO C and X/Open locale categories are:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
```

The *locale* argument is the name of the locale to set the indicated category to. The value of *locale* is implementation-defined.

ERRORS

The ToolTalk service may return the following error in processing the *Set_Locale* request:

TT_DESKTOP_EINVAL
: The *locale* argument is not valid on the handler's host.

APPLICATION USAGE

The *ttdt_session_join*( ), function can be used to register for, and transparently process, the *Set_Locale* request.

EXAMPLES

The *Set_Locale* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_SET_LOCALE,
                    my_callback);
tt_message_arg_add(msg, TT_IN, Tttk_string, "LC_MONETARY");
tt_message_arg_add(msg, TT_IN, Tttk_string, "de");
tt_message_send(msg);
```

SEE ALSO

setlocale( ) in the , *tt_message_arg_add*( ), *tt_message_send*( ), *ttdt_sender_imprint_on*( ), *ttdt_session_join*( ); *Get_Locale* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Set_Mapped request — set whether a tool is mapped to the screen

SYNOPSIS

```
Set_Mapped(in boolean mapped
           [in messageID commission]);
```

DESCRIPTION

The *Set_Mapped* request sets the mapped state of the optionally specified window, or of the window primarily associated with the handling procid (if no window is specified).

The *mapped* argument is a Boolean value indicating whether the specified window is (to be) mapped to the screen.

The *commission* argument is the ID of the ongoing request, if any, that resulted in the creation of the window in question.

APPLICATION USAGE

The *ttdt_session_join*( ), and *ttdt_message_accept*( ), functions can be used by Xt applications to register for, and transparently process, the *Set_Mapped* request.

EXAMPLES

The *Set_Mapped* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_SET_MAPPED,
                    my_callback);
tt_message_iarg_add(msg, TT_IN, Tttk_boolean, 1);
tt_message_send(msg);
```

SEE ALSO

tt_message_iarg_add( ), *tt_message_send*( ), *ttdt_message_accept*( ), *ttdt_session_join*( ); *Get_Mapped* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Set_Situation request — set a tool's current working directory

SYNOPSIS

```
Set_Situation(in string path);
```

DESCRIPTION

The *Set_Situation* request sets the current working directory.

The *path* argument is the pathname of the working directory that the recipient should use.

APPLICATION USAGE

The *ttdt_session_join*( ), function can be used to register for, and transparently process, the *Set_Situation* request.

EXAMPLES

The *Set_Situation* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_SET_SITUATION,
                    my_callback);
tt_message_arg_add(msg, TT_OUT, Tttk_string, 0);
tt_message_send(msg);
```

SEE ALSO

tt_message_arg_add( ), *tt_message_send*( ), *ttdt_session_join*( ); *Get_Situation* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Signal request — send a (POSIX-style) signal to a tool

SYNOPSIS

```
Signal(in string theSignal);
```

DESCRIPTION

The *Signal* request asks the handling procid to send itself the indicated POSIX signal.

The *theSignal* argument is the signal to send.

APPLICATION USAGE

The *ttdt_session_join*( ), function can be used to register for, and transparently process, the *Signal* request.

EXAMPLES

The *Signal* request can be sent as in the following example:

```
Tt_message msg = tttk_message_create(0, TT_REQUEST, TT_SESSION,
                    the_recipient_procid, TTDT_SIGNAL,
                    my_callback);
tt_message_arg_add(msg, TT_IN, Tttk_string, "SIGHUP");
tt_message_send(msg);
```

SEE ALSO

sigaction( ) in the X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**; *tt_message_arg_add*( ), *tt_message_send*( ), *ttdt_session_join*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

Started notice — a tool has started

SYNOPSIS

```
Started(in string vendor,
        in string toolName,
        in string toolVersion);
```

DESCRIPTION

The *Started* notice announces that a tool has started.

The *vendor* argument is the vendor of the started tool.

The *toolName* argument is the name of the started tool.

The *toolVersion* argument is the version of the started tool.

APPLICATION USAGE

EXAMPLES

A pattern observing the *Started* request can be registered as in the following example:

```
Tt_pattern pat = tt_pattern_create();
tt_pattern_category_set(pat, TT_OBSERVE);
tt_pattern_scope_add(pat, TT_SESSION);
char *ses = tt_default_session();
tt_pattern_session_add(pat, ses);
tt_free(ses);
tt_pattern_op_add(pat, Tttk_Started);
tt_pattern_op_add(pat, Tttk_Stopped);
tt_pattern_callback_add(pat, my_callback);
tt_pattern_register(pat);
```

The *Started* request can be sent with *ttdt_open*( ).

SEE ALSO

tt_free( ), *tt_pattern_callback_add* ( ), *tt_pattern_category_set* ( ), *tt_pattern_op_add* ( ), *tt_pattern_register*( ), *tt_pattern_scope_add* ( ), *tt_pattern_session_add* ( ), *ttdt_open* ( ); *Stopped* notice.

CHANGE HISTORY

First released in Issue 1.

NAME

Status notice — a tool has some status information to announce

SYNOPSIS

```
Status(in string status,
       in string vendor,
       in string toolName,
       in string toolVersion
       [in messageID commission]);
```

DESCRIPTION

The *Status* notice indicates that a tool has status information to announce.

The *status* argument is the status being announced.

The *vendor* argument is the vendor of the sending tool.

The *toolName* argument is the name of the sending tool.

The *toolVersion* argument is the version of the sending tool.

The *commission* argument is the ID of the request, if any, that initiated the operation the status of which is being announced.

APPLICATION USAGE

The *ttdt_subcontract_manage*( ), function can be used to register for, and help process, the *Status* request.

The *Status* request can be sent with *ttdt_message_accept*( ).

The *Status* notice can be used by handlers of requests invoking protracted operations to provide periodic point-to-point status reports to the requester. Doing so has the nice side effect of identifying the handler to the requester, so that the requester can issue a *Quit* request if it wants to.

SEE ALSO

ttdt_message_accept( ), *ttdt_subcontract_manage*( ); *Quit* request.

CHANGE HISTORY

First released in Issue 1.

NAME

Stopped notice — a tool has terminated

SYNOPSIS

```
Stopped(in string vendor,
        in string toolName,
        in string toolVersion);
```

DESCRIPTION

The *Stopped* notice announces that a tool has exited.

The *vendor* argument is the vendor of the terminated tool.

The *toolName* argument is the name of the terminated tool.

The *toolVersion* argument is the version of the terminated tool.

EXAMPLES

A pattern observing the *Stopped* request can be registered as in the following example:

```
Tt_pattern pat = tt_pattern_create();
tt_pattern_category_set(pat, TT_OBSERVE);
tt_pattern_scope_add(pat, TT_SESSION);
char *ses = tt_default_session();
tt_pattern_session_add(pat, ses);
tt_free(ses);
tt_pattern_op_add(pat, Tttk_Started);
tt_pattern_op_add(pat, Tttk_Stopped);
tt_pattern_callback_add(pat, my_callback);
tt_pattern_register(pat);
```

The *Stopped* request can be sent with *ttdt_close*( ).

SEE ALSO

tt_free( ), *tt_pattern_callback_add* ( ), *tt_pattern_category_set* ( ), *tt_pattern_op_add* ( ), *tt_pattern_register*( ), *tt_pattern_scope_add* ( ), *tt_pattern_session_add* ( ), *ttdt_close*( ); *Started* notice.

CHANGE HISTORY

First released in Issue 1.

6.6.2 Media Exchange Message Set

The *Media* conventions allow a tool to be a container for arbitrary media, or to be a media player/editor that can be driven from such a container. These conventions allow a container application to compose, display, edit, print or transform a document of an arbitrary media type, without understanding anything about the format of that media type. The ToolTalk service routes container requests to the user's preferred tool for the given media type and operation. This includes routing the request to an already-running instance of the tool if that instance is best-positioned to handle the request.

The following types and argument names are used in message **SYNOPSIS** descriptions:

boolean
A vtype for logical values. The underlying data type of boolean is integer; that is, arguments of this vtype should be manipulated with *tt_*_arg_ival*[*_set*]( ) and *tt_*_iarg_add( )* functions. Zero means false; non-zero means true.

string
A vtype for character strings. Arguments of this vtype should be manipulated with *tt_*_arg_val*[*_set*]( ) and *tt_*_arg_add*( ) functions.

bytes A vtype for character strings that can include null characters.

messageID
A vtype for uniquely identifying messages. The underlying data type of **messageID** is **string**. The **messageID** of a **Tt_message** is returned by *tt_message_id*( ).

title A vtype for character strings intended to be used for document names or titles.

mediaType
The name of a media format. The media type of a document allows messages about that document to be dispatched to the appropriate tool. XCDE conforming systems support at least the media types in Section 3.1 on page 25.

NAME

Deposit request — save a document to its backing store

SYNOPSIS

```
[file] Deposit(in mediaType contents
               [in messageID commission]);
```

DESCRIPTION

The *Deposit* request saves a document to its backing store. This request is different from the *Save* request in that the requester (not the handler) has the data to be saved.

The *contents* argument is the contents of the document. If this argument is unset (in other words, has a value of **(char *)0),** then the contents of the document are in the file named in the message's *file* attribute. The data type (*mediaType*) of the *contents* argument should be **string**, unless nulls are valid in the given media type, in which case the data type must be **bytes**.

The *commission* argument contains the message ID of the *Edit* request that caused the creation of this buffer.

APPLICATION USAGE

The *ttmedia_load*( ) function can be used to register for, and help process, this message.

This message can be sent with the *ttmedia_Deposit*( ) function.

The *Deposit* request is useful for cases where the user may perform an intermediate save of modifications to a document that is the subject of an *Edit* or *Display* request in progress. In the latter case, the *Deposit* may fail on a TT_DESKTOP_EACCES error if the handler does not allow updates to the document being displayed.

Handlers receiving this request should reply before deleting any file named in the message's *file* attribute, but this is optional and applications should not rely on this.

ERRORS

The ToolTalk service may return one of the following errors in processing the *Deposit* request:

TT_DESKTOP_EACCES
: The document is read-only.

TT_DESKTOP_ENOENT
: The file that was alleged to contain the document does not exist.

TT_DESKTOP_ENODATA
: The in-mode *contents* argument had no value and the *file* attribute of the message was not set.

TT_MEDIA_ERR_FORMAT
: The document is not a valid instance of the media type.

SEE ALSO

ttmedia_load( ), *ttmedia_Deposit*( ); *Intro*, *Display*, *Edit*, *Status* requests.

CHANGE HISTORY

First released in Issue 1.

NAME

Display request — display a document

SYNOPSIS

```
[file] Display(in mediaType contents
               [in title docName]);
```

DESCRIPTION

The *Display* request causes the handler to display (present or manifest) a document to the user. For example, an audio manipulation utility would be said to ''display'' audio documents when it plays them.

The handler must decide issues such as:

- When the display operation can be deemed completed
- What user gesture signals the completion of the display
- What the handling tool should do with itself after replying

The *contents* argument is the contents of the document. If this argument is unset (in other words, has a value of **(char *)0),** then the contents of the document are in the file named in the message's *file* attribute. The data type (*mediaType*) of the *contents* argument should be **string**, unless nulls are valid in the given media type, in which case the data type must be **bytes**.

The *docName* argument contains the name of the document. If the *docName* argument is absent and the *file* attribute is set, the file name is considered to be the title of the document. This string would be suitable for display in a window title bar, for example.

APPLICATION USAGE

The *ttmedia_ptype_declare*( ) function can be used to register for, and help process, this message.

This message can be sent with the *ttmedia_load*( ) function.

When the document to be displayed is read-only or unlikely to be modified the *Display* message is frequently used instead of the *Edit* message.

EXAMPLES

To display a PostScript document, the application can send a *Display* request with a first argument whose vtype is **PostScript**, and whose value is a vector of bytes such as:

```
%!\n/inch {72 mul} def...
```

The **\n** in the example represents the newline character. The notation is the same as in the ISO C standard.

To display a PostScript document contained in a file, the application can send a *Display* request with the *file* attribute set to that file and with an unset first argument whose vtype is **PostScript**.

ERRORS

The ToolTalk service may return one of the following errors in processing the *Display* request:

TT_DESKTOP_ENOENT
: The file that was alleged to contain the document does not exist.

TT_DESKTOP_ENODATA
: The in-mode *contents* argument had no value and the *file* attribute of the message was not set.

TT_MEDIA_ERR_FORMAT
: The document is not a valid instance of the media type.

SEE ALSO

ttmedia_ptype_declare( ), *ttmedia_load*( ); *Intro*, *Deposit*, *Edit*, *Status* requests.

CHANGE HISTORY

First released in Issue 1.

NAME

Edit request — compose or edit a document

SYNOPSIS

```
[file] Edit([out|inout] mediaType contents
            [in title docName]);
```

DESCRIPTION

The *Edit* request causes the handler to edit a document and reply with the new contents when the editing is completed.

It is up to the handler to decide issues such as:

- When the editing operation can be deemed completed
- What user gesture signals the completion of the editing
- What the handling tool should do with itself after replying

If the handling tool supports some form of intermediate save operation during editing, it must send a *Deposit* request back to the tool that requested the *Edit*.

The *contents* argument is the contents of the document. If this argument is unset (in other words, has a value of **(char *)0),** then the contents of the document are in the file named in the message's *file* attribute. The data type (*mediaType*) of the *contents* argument should be **string**, unless nulls are valid in the given media type, in which case the data type must be **bytes**.

If the *contents* argument is of mode **out**, then a new document must be composed and its contents returned in this argument.

The *docName* argument contains the name of the document. If the *docName* argument is absent and the *file* attribute is set, the file name is considered to be the title of the document. This string would be suitable for display in a window title bar, for example.

APPLICATION USAGE

The *ttmedia_ptype_declare*( ) function can be used to register for, and help process, this message.

This message can be sent with the *ttmedia_load*( ) function.

EXAMPLES

To edit an X11 XBM bitmap, the application can send an *Edit* request with a first argument whose vtype is **XBM**, and whose value is a string such as:

```
#define foo_width 44\n#define foo_height 94\n
```

The **\n** in the example represents the newline character. The notation is the same as in the ISO C standard.

To edit an X11 XBM bitmap contained in a file, the application can send an *Edit* request naming that file in its *file* attribute, with a first argument whose vtype is **XBM**, and whose value is not set.

ERRORS

The ToolTalk service may return one of the following errors in processing the *Edit* request:

TT_DESKTOP_ECANCELED
: The user overrode the *Edit* request. When an *Edit* request is failed with TT_DESKTOP_ECANCELED, the document should not be updated as a result, but rather should remain as it was before the failure reply was received.

TT_DESKTOP_ENOENT
: The file that was alleged to contain the document does not exist.

TT_DESKTOP_ENODATA
: The in-mode *contents* argument had no value and the *file* attribute of the message was not set.

TT_MEDIA_ERR_FORMAT
: The document is not a valid instance of the media type.

SEE ALSO

ttmedia_ptype_declare( ), *ttmedia_load*( ); *Intro*, *Display* requests.

CHANGE HISTORY

First released in Issue 1.

NAME

Mail request — compose or mail a document

SYNOPSIS

```
[file] Mail(in mediaType contents);

[file] Mail([out|inout] mediaType contents
            [in title docName]);
```

DESCRIPTION

The *Mail* request causes the handler to route a document to a destination using the mail message handling system. The handler is responsible for finding routing information in the document.

When the *contents* argument is of mode **in**, the handler must deliver the document as is, without interacting with the user.

When the *contents* argument is of mode **inout** or **out**, the handler must allow the user to compose or edit the document (and any embedded routing information) before it is delivered. If the handling tool supports some form of intermediate ''save'' operation, it must send a *Deposit* request back to the tool that initiated the *Mail* request.

The *contents* argument is the contents of the document. If this argument is unset (in other words, has a value of **(char *)0),** then the contents of the document are in the file named in the message's *file* attribute. The data type (*mediaType*) of the *contents* argument should be **string**, unless nulls are valid in the given media type, in which case the data type must be **bytes**.

The *docName* argument contains the name of the document. If the *docName* argument is absent and the *file* attribute is set, the file name is considered to be the title of the document. This string would be suitable for display in a window title bar, for example.

APPLICATION USAGE

The *ttmedia_ptype_declare*( ) function can be used to register for, and help process, this message.

This message can be sent with the *ttmedia_load*( ) function.

ERRORS

The ToolTalk service may return one of the following errors in processing the *Mail* request:

TT_DESKTOP_ENOENT
: The file that was alleged to contain the document does not exist.

TT_DESKTOP_ENODATA
: The in-mode *contents* argument had no value and the *file* attribute of the message was not set.

TT_MEDIA_ERR_FORMAT
: The document is not a valid instance of the media type.

SEE ALSO

ttmedia_ptype_declare( ), *ttmedia_load*( ); *Intro*, *Edit* requests.

CHANGE HISTORY

First released in Issue 1.

NAME

Print request — print a document

SYNOPSIS

```
[file] Print(in mediaType contents,
             in boolean inquisitive,
             in boolean covert
             [in title docName]);
```

DESCRIPTION

The *Print* request causes the handler to print a document. The handler must act as if the user had issued, (via the handler's user interface) either a ''Print One'' or ''Print...'' command, depending on the value of the *inquisitive* argument.

The *contents* argument is the contents of the document. If this argument is unset (in other words, has a value of **(char *)0),** then the contents of the document are in the file named in the message's *file* attribute. The data type (*mediaType*) of the *contents* argument should be **string**, unless nulls are valid in the given media type, in which case the data type must be **bytes**.

The *inquisitive* argument is a **boolean** value indicating whether the handler is allowed to block on user input while carrying out the request. However, even if *inquisitive* is True, the handler is not required to seek such input.

The *covert* argument is a **boolean** value indicating whether the handler may make itself apparent to the user as it carries out the request. If False, the recipient need not make itself apparent.

If both the *inquisitive* argument and the *covert* argument are True, the recipient should attempt to limit its presence to the minimum needed to receive any user input desired; for example, through iconification.

The *docName* argument contains the name of the document. If the *docName* argument is absent and the *file* attribute is set, the file name is considered to be the title of the document. This string would be suitable for display in a window title bar, for example.

APPLICATION USAGE

The *ttmedia_ptype_declare*( ) function can be used to register for, and help process, this message.

This message can be sent with the *ttmedia_load*( ) function.

EXAMPLES

To print a PostScript document, the application can send a request of the form:

```
Print(in PostScript contents,
      in boolean inquisitive,
      in boolean covert);
```

with a first argument whose value is a vector of bytes such as:

```
%!\n/inch {72 mul} def...
```

The **\n** in the example represents the newline character. The notation is the same as in the ISO C standard.

To print a PostScript document contained in a file, the application can send the *Print* request as above, with the *file* attribute set to the relevant file and with the value of the first argument not set.

ERRORS

The ToolTalk service may return one of the following errors in processing the *Print* request:

TT_DESKTOP_ENOENT
: The file that was alleged to contain the document does not exist.

TT_DESKTOP_ENODATA
: The in-mode *contents* argument had no value and the *file* attribute of the message was not set.

TT_MEDIA_ERR_FORMAT
: The document is not a valid instance of the media type.

SEE ALSO

ttmedia_ptype_declare( ), *ttmedia_load*( ); *Intro*, *Status* requests.

CHANGE HISTORY

First released in Issue 1.

NAME

Translate request — translate a document from one media type to another

SYNOPSIS

```
[file] Translate(in mediaType contents,
                 out mediaType output,
                 in boolean inquisitive,
                 in boolean covert
                 [in messageID counterfoil]);
```

DESCRIPTION

The *Translate* request causes the handler to translate a document from one media type to another and return the translation. The translation must be the best possible representation of the document in the target media type, even if the resulting representation cannot be exactly translated back into the original document.

The *contents* argument is the contents of the document. If this argument is unset (in other words, has a value of **(char *)0),** then the contents of the document are in the file named in the message's *file* attribute. The data type (*mediaType*) of the *contents* argument should be **string**, unless nulls are valid in the given media type, in which case the data type must be **bytes**.

The *output* argument is the translation of the document.

The *inquisitive* argument is a **boolean** value indicating whether the handler is allowed to block on user input while carrying out the request. However, even if *inquisitive* is True, the handler is not required to seek such input.

The *covert* argument is a **boolean** value indicating whether the handler may make itself apparent to the user as it carries out the request. If False, the recipient need not make itself apparent.

If both the *inquisitive* argument and the *covert* argument are True, the recipient should attempt to limit its presence to the minimum needed to receive any user input desired; for example, through iconification.

The *counterfoil* argument is a unique string created by the message sender to give both sender and receiver a way to refer to this request in other correspondence. Typically this string is created by concatenating a process ID and a counter. This argument should be included if the sender anticipates a need to communicate with the handler about this request before it is completed; for example, to cancel it. When this argument is included, and the handler determines that an immediate reply is not possible, the handler must immediately send at least one *Status* notice point-to-point back to the requester, so as to identify itself to the requester.

APPLICATION USAGE

To provide a speech-to-text service, a tool can handle requests of the form:

```
Translate(in Sun_Audio contents,
          out ISO_Latin_1 output,
          ...);
```

To provide an OCR (optical character recognition) service, a tool can handle requests of the form:

```
Translate(in GIF contents,
          out ISO_Latin_1 output,
          ...);
```

ERRORS

The ToolTalk service may return one of the following errors in processing the *Translate* request:

TT_DESKTOP_ENOENT
: The file that was alleged to contain the document does not exist.

TT_DESKTOP_ENODATA
: The in-mode *contents* argument had no value and the *file* attribute of the message was not set.

TT_MEDIA_ERR_FORMAT
: The document is not a valid instance of the media type.

SEE ALSO

Intro, *Abstract*, *Interpret*, *Status* requests.

CHANGE HISTORY

First released in Issue 1.

Chapter 7

Drag and Drop

7.1 Introduction

The XCDE drag and drop services defined in this chapter are an extension of the Motif drag and drop services defined in the X/Open CAE Specification, **Motif Toolkit API**. Convenience APIs are included that reduce programming complexity for common operations, such as the dragging of files.

7.2 Functions

This section defines the functions, macros and external variables that provide XCDE drag and drop services to support application portability at the C-language source level.

NAME

DtDndCreateSourceIcon — create a drag source icon

SYNOPSIS

```
#include <Dt/Dnd.h>

Widget DtDndCreateSourceIcon(Widget parent,
                             Pixmap pixmap,
                             Pixmap mask);
```

DESCRIPTION

The *DtDndCreateSourceIcon*( ) function creates a Motif drag icon, named **sourceIcon**, based on the characteristics of the *pixmap* argument. The resulting drag icon is suitable for use with *DtDndDragStart*( ).

The *parent* argument is the parent of the drag icon. Typically this widget is the drag source.

The *pixmap* argument is the pixmap representation of the data to be dragged.

The *mask* argument is the mask for the *pixmap*.

When it calls *XmCreateDragIcon*( ), the *DtDndCreateSourceIcon*( ) function sets Motif resources in the drag icon; the application must not modify the values of any of these resources:

XmNwidth
XmNheight
XmNpixmap
XmNmask
XmNdepth

RETURN VALUE

Upon successful completion, the *DtDndCreateSourceIcon*( ) function returns a drag icon created by calling *XmCreateDragIcon*( ) with the characteristics of the *pixmap*; otherwise, it returns NULL.

SEE ALSO

<Dt/Dnd.h>, *DtDndDragStart*( ); *XmCreateDragIcon*( ), *XmDragIcon*( ), *XmDragStart*( ) in the X/Open CAE Specification, **Motif Toolkit API**; *XtDestroyWidget*( ) in the X/Open CAE Specification, **Window Management**: **X Toolkit Intrinsics**.

CHANGE HISTORY

First released in Issue 1.

NAME

DtDndDragStart, DtDndVaDragStart — initiate a drag

SYNOPSIS

```
#include <Dt/Dnd.h>

Widget DtDndDragStart(Widget dragSource,
                      XEvent *event,
                      DtDndProtocol protocol,
                      Cardinal numItems,
                      unsigned char operations,
                      XtCallbackList convertCallback,
                      XtCallbackList dragFinishCallback,
                      ArgList argList,
                      Cardinal argCount);

Widget DtDndVaDragStart(Widget dragSource,
                        XEvent *event,
                        DtDndProtocol protocol,
                        Cardinal numItems,
                        unsigned char operations,
                        XtCallbackList convertCallback,
                        XtCallbackList dragFinishCallback,
                        ...);
```

DESCRIPTION

The *DtDndDragStart*( ) and *DtDndVaDragStart*( ) functions initiate a Motif drag, with drag visuals appropriate to the type of data being dragged, and updates the translation table of the drag context. Either of the functions is called from the application's event handler, which interprets mouse events to determine when a drag should begin.

The only difference between *DtDndDragStart*( ) and *DtDndVaDragStart*( ) is how the argument list is passed. The argument list is passed as an *ArgList* to *DtDndDragStart*( ) and using *varargs* for *DtDndVaDragStart*( ).

The *dragSource* argument is the widget that received the event that triggered the drag.

The *event* argument is the button press or button motion event that triggered the drag.

The *protocol* argument specifies the protocol used for the data transfer. Valid values are:

DtDND_TEXT_TRANSFER
: A list of text is being transferred.

DtDND_FILENAME_TRANSFER
: A list of file names is being transferred.

DtDND_BUFFER_TRANSFER
: A list of memory buffers is being transferred.

The *numItems* argument specifies the number of items being dragged.

The *operations* argument indicates which operations the *dragSource* supports. The operations are:

XmDROP_COPY
: Copy operations are valid.

XmDROP_LINK
: Link operations are valid.

XmDROP_MOVE
Move operations are valid.

A drag source can support any combination of these operations. A combination of operations is specified by the bitwise inclusive OR of several operation values. For example, to support the move and copy operations, the application can specify:

```
XmDROP_MOVE | XmDROP_COPY
```

The *convertCallback* argument is a callback function that is invoked when a drop has started and the drop site has requested data from the drag source. The *convertCallback* is responsible for providing the data that is transferred to the drop site.

The *dragFinishCallback* argument is a callback function that is invoked when the drag and drop transaction is complete. The *dragFinishCallback* is called after the *convertCallback*, if applicable. (The *convertCallback* is called only after a drop has started on an eligible drop site, whereas *dragFinishCallback* is called after the drag finishes, whether or not a drop occurred.) The *dragFinishCallback* should reset any drag motion handler and free any memory allocated by the drag source during the drag and drop transaction.

Argument Value Pairs

The *DtDndDragStart*( ) and *DtDndVaDragStart*( ) functions support the following optional argument-value pairs. Motif resources can be set via the argument list as well, provided they are not resources that are used by the drag and drop subsystem; see **Motif Resources**.

DtNsourceIcon (**Widget**)
Specifies the *XmDragIcon* used to represent the data being dragged. This icon is created using either *DtDndCreateSourceIcon*( ) or *XmCreateDragIcon*( ). If *DtNsourceIcon* is NULL, then a default icon is used, which is appropriate for the data begin dragged. The default value is NULL.

DtNbufferIsText (**Boolean**)
Specifies whether the dragged buffer should also be sourced as text, allowing the buffer to be dropped onto text widgets. This attribute is only valid if *protocol* DtDND_BUFFER_TRANSFER and is ignored for other transfers. If **DtNbufferIsText** is True, the buffer is sourced as text in addition to being sourced as buffers; if it is False, the buffers are sourced only as buffers.

Callbacks

Once the rendezvous with the drop site has been accomplished, the application-provided callback functions are called to perform the transfer of the dragged data.

First, the *convertCallback* is called with a *reason* of DtCR_DND_CONVERT_DATA. The application must set the *DtDndContext* fields appropriate to the transfer protocol to provide the data to be transferred to the drop site.

If the drag operation is XmDROP_MOVE and the drop site requests that the move be completed, the *convertCallback* is called again with a *reason* of DtCR_DND_CONVERT_DELETE. The application should delete its version of the dragged data.

Once the data transfer is complete, the *dragFinishCallback* is called with a *reason* of DtCR_DND_DRAG_FINISH. The application should free any memory allocated in the *convertCallback* and restore any application state.

Callback Information

Each of the callbacks for *DtDndDragStart*( ) and *DtDndVaDragStart*( ) has an associated callback structure. These callbacks cannot be used with the *XtAddCallback*( ) interface.

A pointer to the following structure is passed to the *convertCallback*:

```
typedef struct {
        int reason;
        XEvent *event;
        DtDndContext *dragData;
        DtDndStatus status;
} DtDndConvertCallbackStruct, *DtDndConvertCallback;
```

The *reason* argument indicates why the callback was invoked. The possible reasons for this callback are:

DtCR_DND_CONVERT_DATA
: The callback provides the requested data by setting appropriate fields in the *dragData* structure.

DtCR_DND_CONVERT_DELETE
: The callback completes the XmDROP_MOVE operation by deleting its copy of the dragged data.

The *event* argument points to the *XEvent* that triggered the callback.

The *dragData* argument specifies the *DtDndContext* that contains the data to be dragged. If the *reason* argument is DtCR_DND_CONVERT_DATA, the application must provide the data by setting the relevant fields in the *DtDndContext*, as described in the following **Structures** section. If the *reason* argument is DtCR_DND_CONVERT_DELETE, the application must delete the original data that completes a move operation.

The *status* argument indicates the status of the conversion. The application can set this to DtDND_FAILURE to cancel the conversion and consequently the drag and drop operation. The value of *status* is normally DtDND_SUCCESS.

A pointer to the following structure is passed to the *dragFinishCallback*:

```
typedef struct {
        int reason;
        XEvent *event;
        DtDndContext *dragData;
        Widget sourceIcon;
} DtDndDragFinishCallbackStruct, *DtDndDragFinishCallback;
```

The *reason* argument indicates why the callback was invoked. The valid reason for this callback is DtCR_DND_DRAG_FINISH.

The *event* argument points to the *XEvent* that triggered the callback.

The *sourceIcon* argument specifies the source icon registered in the call to *DtDndDragStart*. This widget is provided to allow the application to free data associated with the source icon and optionally destroy the source icon.

The *dragData* argument specifies the *DtDndContext* that contains the data that was dragged. The application should free any data it associated with the *dragData*.

Structures

The following structures are used by the drag source in the *convertCallback* and *dragFinishCallback* to communicate data between the application and the drag and drop subsystem. The *DtDndContext* structure is passed to these callbacks as *dragData* in the callback structure appropriate to the callback.

In the *convertCallback*, the application that is acting as the drag source is responsible for filling in the *DtDndContext* structure with the data being transferred.

In the *dragFinishCallback*, the application acting as the drag source is responsible for freeing any data it allocated for use in the *DtDndContext* structure.

```
typedef struct _DtDndContext {
        DtDndProtocol protocol;
        int numItems;
        union {
                XmString *strings;
                String *files;
                DtDndBuffer *buffers;
        } data;
} DtDndContext;
```

The *protocol* argument indicates the data transfer protocol under which the data in the *DtDndContext* is being transferred. Valid values are:

DtDND_TEXT_TRANSFER
: A list of text is being transferred.

DtDND_FILENAME_TRANSFER
: A list of file names is being transferred.

DtDND_BUFFER_TRANSFER
: A list of memory buffers is being transferred.

The *numItems* argument indicates the number of items being transferred.

The *data* argument is a union containing data that should be stored and read in the format corresponding to the specified *protocol*. The data structures corresponding to the transfer protocols are as follows.

The *strings* argument is valid if the *protocol* is DtDND_TEXT_TRANSFER. The *strings* argument is an array of pointers to Motif strings containing the text being transferred.

The *files* argument is valid if the *protocol* is DtDND_FILENAME_TRANSFER. It is an array of pointers to the names of the files being transferred. The file names have been converted to local host file names where possible.

The *buffers* argument is valid if the *protocol* is DtDND_BUFFER_TRANSFER. It is an array of pointers to *DtDndBuffer* structures containing the meory buffers being transferred.

The following structure is used with DtDND_FILENAME_TRANSFER:

```
typedef struct _DtDndBuffer {
        void *bp;
        int size;
        string name;
} DtDndBuffer;
```

The *bp* argument points to the buffer data being transferred.

The *size* argument indicates the number of bytes in the buffer.

The *name* argument points to the name of the buffer.

Motif Resources

When it calls *XmDragStart*( ), the *DtDndDragStart*( ) function sets Motif resources; the application must not modify the values of any of these resources. Resources other than those listed here can be used and are passed through to the underlying *XmDragStart*( ) call.

The following resources are modified by *DtDndDragStart*( ) and *DtDndVaDragStart*( ) in the Motif Drag Context:

XmNblendModel
XmNclientData
XmNconvertProc
XmNcursorBackground
XmNcursorForeground
XmNdragDropFinishCallback
XmNdragOperations
XmNdropFinishCallback
XmNdropStartCallback
XmNexportTargets
XmNnumExportTargets
XmNsourcePixmapIcon
XmNtopLevelEnterCallback

The following resources are modified by *DtDndDragStart*( ) and *DtDndVaDragStart*( ) in the Motif Drag Icon:

XmNattachment
XmNdepth
XmNheight
XmNhotX
XmNhotY
XmNmask
XmNoffsetX
XmNoffsetY
XmNpixmap
XmNwidth

RETURN VALUE

Upon successful completion, the *DtDndDragStart*( ) function returns the drag context widget created when the Motif drag is started; otherwise, it returns NULL if the drag could not be started.

SEE ALSO

<Dt/Dnd.h>, *DtDtsFileToDataType*( ), *DtDndCreateSourceIcon*( ), *DtDndDropRegister*( ), *DtDndVaDropRegister*( ), *DtDndDropUnregister*( ); *XmCreateDragIcon*( ), *XmDragContext*( ), *XmDragIcon*( ), *XmDragStart*( ) in the X/Open CAE Specification, **Motif Toolkit API**.

CHANGE HISTORY

First released in Issue 1.

NAME

DtDndDropRegister, DtDndVaDropRegister — specify a drop site

SYNOPSIS

```
#include <Dt/Dnd.h>

void DtDndDropRegister(Widget dropSite,
                       DtDndProtocol protocols,
                       unsigned char operations,
                       XtCallbackList transferCallback,
                       ArgList argList,
                       Cardinal argCount);

void DtDndVaDropRegister(Widget dropSite,
                         DtDndProtocol protocols,
                         unsigned char operations,
                         XtCallbackList transferCallback,
                         ...);
```

DESCRIPTION

The *DtDndDropRegister*( ) and *DtDndVaDropRegister*( ) functions register a Motif drop site with import targets based on the specified data transfer protocols. *DtDndDropRegister*( ) may be called to register a widget as a drop site at any time, typically soon after the widget is created.

The only difference between *DtDndDropRegister*( ) and *DtDndVaDropRegister*( ) is how the argument list is passed. The argument list is passed as an *ArgList* to *DtDndDropRegister*( ) and using *VarArgs* for *DtDndVaDropRegister*( ).

The *dropSite* argument specifies the widget to register as the drop site.

The *protocol* argument specifies the set of data transfer protocols in which the drop site is able to participate. Valid values are:

DtDND_TEXT_TRANSFER
: The drop site can transfer a list of text.

DtDND_FILENAME_TRANSFER
: The drop site can transfer a list of file names.

DtDND_BUFFER_TRANSFER
: The drop site can transfer a list of memory buffers.

A drop site can support any combination of these protocols. A combination of protocols is specified by the bitwise inclusive OR of several protocol values.

The *operations* argument specifies the set of valid operations associated with a drop site. The operations are:

XmDROP_COPY
: Copy operations are valid. The data will be copied from the drag source.

XmDROP_LINK
: Link operations are valid. The data will be linked using an alternative method.

XmDROP_MOVE
: Move operations are valid. The data will be copied, and optionally deleted, from the drag source.

A drop site can support any combination of these operations. A combination of operations is specified by the bitwise inclusive OR of several operation values.

The *transferCallback* argument specifies the callback to be called when the dropped data object has been received by the drop site. The *transferCallback* is responsible for transferring the data from the *dropData* to the appropriate internal data structures at the drop site.

The *argList* argument specifies the optional argument list.

The *argCount* argument specifies the number of arguments in *argList*.

Argument Value Pairs

The *DtDndDragStart*( ) and *DtDndVaDragStart*( ) functions support the following optional argument-value pairs. Motif resources can be set via the argument list as well, provided they are not resources that are used by the drag and drop subsystem; see **Motif Resources**.

DtNregisterChildren (**Boolean**)
: Specifies whether children of a composite drop site widget should be registered. If True, then the composite *dropSite* widget and its children are registered as a single drop site. If False, then only the *dropSite* widget itself is registered as the drop site. The default is False.

DtNtextIsBuffer (**Boolean**)
: Specifies whether the drops of text selections should be treated as buffer drops. This attribute is only valid if *protocols* includes DtDND_BUFFER_TRANSFER. If **DtNtextIsBuffer** is True, text drops are accepted as unnamed buffers; if it is False, only drops of the specified *protocols* are accepted. The default is False.

DtNpreserveRegistration (**Boolean**)
: Specifies whether to preserve any existing drop site registration for the *dropSite* widget. The application can disable preserving the drop site registration if the *dropSite* widget is known not to be registered as a drop site or that registration is not desired. This may improve drop site registration performance. If **DtNpreserveRegistration** is True, existing drop site registration is preserved; if it is False, the existing drop site registration is replaced. The default is True.

DtNdropAnimateCallback (**XtCallbackList**)
: Specifies the callback to be called when the drop is complete. This enables graphical animation upon successful completion of a drop. This callback is called after the *transferCallback* is called and after Motif performs the ‘‘melt’’ effect. The Motif Drag Context is in the process of being destroyed at this point so the application must not use it or any of its values in the *dropAnimateCallback*. The default is NULL.

Callbacks

Once the rendezvous with the drag source has been accomplished, the application-provided callback functions are called to perform the data transfer.

First, the *transferCallback* is called with a *reason* of DtCR_DND_TRANSFER_DATA. The application should access the *DtDndContext* fields appropriate for the transfer protocol. The application should parse or type the dropped data to determine whether it is acceptable. If the dropped data is not acceptable, the status field of the *DtDndTransferCallbackStruct* should be set to DtDND_FAILURE.

If the **DtNdropAnimateCallback** attribute has been specified, the *dropAnimateCallback* is then called with a *reason* of DtCR_DND_DROP_ANIMATE. The application should perform any application-provided animations for the drop.

When the *transferCallback* (or the *dropAnimateCallback*, if specified) returns, all memory associated with the drop transfer is freed. Any data in the callback structures that will be used after the drop transfer is complete must be copied by the application.

Callback Information

Each of the callbacks for *DtDndDropRegister*( ) and *DtDndVaDropRegister*( ) has an associated callback structure. These callbacks cannot be used with the *XtAddCallback*( ) interface.

A pointer to the following structure is passed to the *transferCallback*:

```
typedef struct {
        int reason;
        XEvent *event;
        Position x, y;
        unsigned char operation;
        DtDndContext *dropData;
        Boolean completeMove;
        DtDndStatus status;
} DtDndTransferCallbackStruct, *DtDndTransferCallback;
```

The *reason* argument indicates why the callback was invoked: DtCR_DND_TRANSFER_DATA.

The *event* argument is always set to NULL by Motif drag and drop.

The *x* and *y* arguments indicate the coordinates of the dropped item in relation to the origin of the drop site widget.

The *operation* argument indicates the type of drop: XmDROP_COPY, XmDROP_MOVE or XmDROP_LINK.

The *dropData* argument contains the data that has been dropped.

The *dragContext* argument specifies the ID of the Motif Drag Context widget associated with this drag and drop transaction.

The *completeMove* argument indicates whether a move operation needs to be completed. If the *operation* is XmDROP_MOVE and *completeMove* is set to False in the *transferCallback*, a delete does not occur. By default, *completeMove* is True and a delete occurs to complete the move operation. The *completeMove* field should be set to False if an alternative method will be used to complete the move.

The *status* argument indicates the success or failure of the data transfer. If the data could not be appropriately moved, copied or linked, the *status* field must be set to DtDND_FAILURE. By default, the *status* field is set to DtDND_SUCCESS.

A pointer to the following structure is passed to the *dropAnimateCallback*:

```
typedef struct {
        int reason;
        XEvent *event;
        Position x, y;
        unsigned char operation;
        DtDndContext *dropData;
} DtDndDropAnimateCallbackStruct, *DtDndDropAnimateCallback;
```

The *reason* argument indicates why the callback was invoked. The valid reason is DtCR_DND_DROP_ANIMATE.

The *event* argument is always set to NULL by Motif drag and drop.

The *x* and *y* arguments indicate the coordinates of the dropped item in relation to the origin of the drop site widget.

The *operation* argument indicates the type of drop: XmDROP_COPY, XmDROP_MOVE or XmDROP_LINK.

The *dropData* argument contains the data that has been dropped.

Structures

The following structures are used by the drop site in the *transferCallback* to get the transferred data from the drag and drop subsystem. The *DtDndContext* structure is passed as *dropData* in the *DtDndTransferCallbackStruct* structure.

```
typedef struct {
        DtDndProtocol protocol;
        int numItems;
        union {
                XmString *strings;
                String *files;
                DtDndBuffer *buffers;
        } data;
} DtDndContext;
```

The *protocol* argument indicates the data transfer protocol under which the data in the *DtDndContext* is being transferred. Valid values are:

DtDND_TEXT_TRANSFER
: A list of text is being transferred.

DtDND_FILENAME_TRANSFER
: A list of file names is being transferred.

DtDND_BUFFER_TRANSFER
: A list of memory buffers is being transferred.

The *numItems* argument indicates the number of items being transferred.

The *data* argument is a union containing data that the drop site should access in the format corresponding to the specified *protocol*. The data structures corresponding to the transfer protocols are as follows.

The *strings* argument is valid if the *protocol* is DtDND_TEXT_TRANSFER. The *strings* argument is an array of pointers to Motif strings containing the text being transferred.

The *files* argument is valid if the *protocol* is DtDND_FILENAME_TRANSFER. It is an array of pointers to the names of the files being transferred. The file names have been converted to local host file names where possible.

The *buffers* argument is valid if the *protocol* is DtDND_BUFFER_TRANSFER. It is an array of pointers to *DtDndBuffer* structures containing the meory buffers being transferred.

The following structure is used with DtDND_FILENAME_TRANSFER:

```
typedef struct _DtDndBuffer {
        void *bp;
        int size;
        string name;
} DtDndBuffer;
```

The *bp* argument points to the buffer data being transferred.

The *size* argument indicates the number of bytes in the buffer.

The *name* argument points to the name of the buffer.

Motif Resources

When it calls *XmDropSiteRegister*( ), the *DtDndDropRegister*( ) and *DtDndVaDropRegister*( ) functions set Motif resources; the application must not modify the values of any of these resources. Resources other than those listed here can be used and are passed through to the underlying *XmDropRegister*( ). call.

The following resources are modified by *DtDndDropRegister*( ) and *DtDndVaDropRegister*( ) in the Motif Drag Context.

XmNdestroyCallback
XmNdropTransfers
XmNnumDropTransfers
XmNtransferProc
XmNtransferStatus

The following resources are modified by *DtDndDropRegister*( ) and *DtDndVaDropRegister*( ) in the Motif Drop Site.

XmNdropProc
XmNdropSiteOperations
XmNdropSiteType
XmNimportTargets
XmNnumImportTargets

RETURN VALUE

The *DtDndDropRegister*( ) and *DtDndVaDropRegister*( ) functions return no value.

SEE ALSO

<Dt/Dnd.h>, *DtDtsDataTypeToAttributeValue*( ), *DtDndDragStart*( ), *DtDndVaDragStart*( ), *DtDndDropUnregister*( ); *XmDragContext*( ), *XmDropSite*( ), *XmDropSiteRegister*( ), *XmDropSiteUpdate*( ), *XmDropTransfer*( ), *XmDropTransferStart*( ) in the X/Open CAE Specification, **Motif Toolkit API**.

CHANGE HISTORY

First released in Issue 1.

7.3 Headers

This section describes the contents of headers used by the XCDE drag and drop functions, macros and external variables.

Headers contain the definition of symbolic constants, common structures, preprocessor macros and defined types. Each function in Section 7.2 specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

NAME

Dt/Dnd.h — Drag and drop definitions

SYNOPSIS

```
#include <Dt/Dnd.h>
```

DESCRIPTION

The **<Dt/Dnd.h>** header defines the following enumeration types:

```
DtCR_DND_CONVERT_DATA
DtCR_DND_CONVERT_DELETE
DtCR_DND_DRAG_FINISH
DtCR_DND_TRANSFER_DATA
DtCR_DND_DROP_ANIMATE
```

The header defines the following enumeration data types, with at least the following members:

DtDndStatus

DtDND_SUCCESS, DtDND_FAILURE

DtDndProtocol

```
DtDND_TEXT_TRANSFER     = (1L << 0)
DtDND_FILENAME_TRANSFER = (1L << 1)
DtDND_BUFFER_TRANSFER   = (1L << 2)
```

The header declares the following structures:

```
typedef struct _DtDndBuffer {
        void *bp;
        int size;
        string name;
} DtDndBuffer;

typedef struct _DtDndContext {
        DtDndProtocol protocol;
        int numItems;
        union {
                XmString *strings;
                String *files;
                DtDndBuffer *buffers;
        } data;
} DtDndContext;

typedef struct _DtDndConvertCallbackStruct {
        int reason;
        XEvent *event;
        DtDndContext *dragData;
        DtDndStatus status;
} DtDndConvertCallbackStruct, *DtDndConvertCallback;

typedef struct _DtDndDragFinishCallbackStruct {
        int reason;
        XEvent *event;
        DtDndContext *dragData;
        Widget sourceIcon;
} DtDndDragFinishCallbackStruct, *DtDndDragFinishCallback;
```

```
typedef struct _DtDndTransferCallbackStruct {
       int reason;
       XEvent *event;
       Position x, y;
       unsigned char operation;
       DtDndContext *dropData;
       Boolean completeMove;
       DtDndStatus status;
} DtDndTransferCallbackStruct, *DtDndTransferCallback;

typedef struct _DtDndDropAnimateCallbackStruct {
       int reason;
       XEvent *event;
       Position x, y;
       unsigned char operation;
       DtDndContext *dropData;
} DtDndDropAnimateCallbackStruct, *DtDndDropAnimateCallback;
```

The header declares the following as functions:

```
Widget DtDndCreateSourceIcon(Widget parent,
                             Pixmap pixmap,
                             Pixmap mask);

Widget DtDndDragStart(Widget dragSource,
                      XEvent *event,
                      DtDndProtocol protocol,
                      Cardinal numItems,
                      unsigned char operations,
                      XtCallbackList convertCallback,
                      XtCallbackList dragFinishCallback,
                      ArgList argList,
                      Cardinal argCount);

Widget DtDndVaDragStart(Widget dragSource,
                        XEvent *event,
                        DtDndProtocol protocol,
                        Cardinal numItems,
                        unsigned char operations,
                        XtCallbackList convertCallback,
                        XtCallbackList dragFinishCallback,
                        ...);

void DtDndDropRegister(Widget dropSite,
                       DtDndProtocol protocols,
                       unsigned char operations,
                       XtCallbackList transferCallback,
                       ArgList argList,
                       Cardinal argCount);
```

```
void DtDndVaDropRegister(Widget dropSite,
                         DtDndProtocol protocols,
                         unsigned char operations,
                         XtCallbackList transferCallback,
                         ...);

void DtDndDropUnregister(Widget dropSite);
```

CHANGE HISTORY

First released in Issue 1.

7.4 Protocols

The drag and drop protocols provide policy for matching and data transfer between the drag initiator and the drop receiver of file names, selected text spans and application-defined structured data formats.

The drag and drop protocols use the standard X11 selection targets, where available, with the addition of several new selection targets where required.

These protocols provide for the transfer of the following types of data:

- Selected Text
- File Names
- Buffers

7.4.1 Protocol Overview

Each protocol consists of the following:

7.4.1.1 Drag and Drop API Protocol

Each protocol described corresponds to a specific **DtDndProtocol** enumeration value.

7.4.1.2 Export/Import Targets

The Motif drag and drop API provides support for matching of the data transfer protocol between the drag initiator and the various drop receivers. This allows the user to determine readily which drop sites will accept the dragged data.

The drag initiator sets the **XmNexportTargets** resource of the **XmDragContext** to the list of target atoms that describe the data being dragged. The drop receiver sets the **XmNimportTargets** resource of the **XmDropSite** to the list of target atoms that describe the data that it will accept. The Motif drag and drop subsystem allows drops when the **XmNexportTargets** and **XmNimportTargets** have at least one target in common.

7.4.1.3 Data Transfer Protocol

Once the drag initiator has dropped on the drop receiver, the transfer of data is begun. The transfer is accomplished using X selections and is controlled by the drop receiver.

The drop receiver starts all transfers by converting the selection into the ICCCM TARGETS target to get the set of available selection targets. (See the X/Open CAE Specification, **Window Management: File Formats and Application Conventions** for a description of converting targets.) It then chooses the appropriate selections from that set and requests that the drag initiator convert each requested selection. Each protocol has a set of selection targets that are used to transfer all the necessary data. These target conversions are usually initiated by calling *XmDropTransferStart*( ).

7.4.1.4 Move Completion

When the operation of the drop is XmDROP_MOVE, the drop receiver must complete the move using an appropriate method. For most data transfers, this is accomplished by converting the selection into the ICCCM DELETE target to tell the drag initiator that it may delete the data. For most file name transfers, this is accomplished via the file system.

7.4.2 Text Transfer Protocol

The text transfer protocol is used to exchange text selections.

7.4.2.1 Drag and Drop API

This is the protocol used when a **DtDndProtocol** of DtDND_TEXT_TRANSFER is specified.

7.4.2.2 Export/Import Targets

The export or import targets are any of the following; the target describing the character encoding of the text selection, COMPOUND_TEXT, STRING or TEXT.

7.4.2.3 Data Transfer Protocol

The transfer of text selections follows the protocols described in the ICCCM section of the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**. If the character encoding of the drag initiator and drop receiver are the same, that target should be converted to get the text selection. If the character encoding are different, the drop receiver should attempt to convert the standard text targets in the following order: COMPOUND_TEXT, STRING or TEXT.

7.4.2.4 Move Completion

The move is completed by converting the selection into the ICCCM DELETE target.

7.4.3 File Name Transfer Protocol

The transfer protocol is used to exchange file names.

7.4.3.1 Drag and Drop API

This is the protocol used when a **DtDndProtocol** of DtDND_FILENAME_TRANSFER is specified.

7.4.3.2 Export/Import Targets

The export or import targets are FILE_NAME and, optionally, _DT_NETFILE if capable of providing the file name in network canonical form using *tt_file_netfile*( ) and *tt_netfile_file*( ).

7.4.3.3 Data Transfer Protocol

If the ICCCM HOST_NAME target is in the list of target atoms, it is converted. If the returned host name is different than the host name for the drop receiver and the _DT_NETFILE target is in the list of target atoms, it is converted. The drag initiator uses *tt_file_netfile*( ) to encode the file names and the drop receiver uses *tt_netfile_file*( ) to decode the file names.

If the hosts are the same for both the drag initiator and the drop receiver or either the HOST_NAME or the _DT_NETFILE targets are not in the list of target atoms from the drag initiator, the drop receiver converts the ICCCM FILE_NAME target. No encoding of the file names occurs in this case.

7.4.3.4 Move Completion

Moves of file names can be accomplished atomically using standard file system operations. Drop receivers are encouraged to use the file system. The drop receiver may alternatively choose to use the ICCCM DELETE target to complete the XmDROP_MOVE and the drag initiator must be ready to comply.

7.4.4 Buffer Transfer Protocol

The transfer protocol is used to exchange memory buffers.

7.4.4.1 Drag and Drop API

This is the protocol used when a **DtDndProtocol** of DtDND_BUFFER_TRANSFER is specified.

7.4.4.2 Export/Import Targets

The export and import targets are _DT_BUFFER_DATA, _DT_BUFFER_LENGTHS and, optionally, _DT_BUFFER_NAMES.

7.4.4.3 Data Transfer Protocol

The _DT_BUFFER_DATA and _DT_BUFFER_LENGTHS targets are converted to transfer the buffer data.

The data of the buffers is encoded into the _DT_BUFFER_DATA target as an array of bytes. The lengths in bytes of each buffer are encoded into _DT_BUFFER_LENGTHS. Each length is used to index into the _DT_BUFFER_DATA array.

If the _DT_BUFFER_NAMES target is available, it is converted to transfer the names of the buffers.

7.4.4.4 Move Completion

The move is completed by converting the selection into the ICCCM DELETE target.

7.4.5 Selection Targets

The following table describes the selection targets used in the drag and drop data matching and transfer protocols.

Atom	Type	Description
TARGETS	ATOM	A list of valid target atoms. See section 3.6.2 of the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**.
DELETE	NULL	Used to delete the dropped data. If the drop receiver wishes to perform a move operation on the data, after copying the data it should request conversion of the DELETE target. See section 3.6.3 of the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**.
COMPOUND_TEXT	COMPOUND_TEXT	The text selection in compound text format. See section 3.7.1 of the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**.
STRING	STRING	The text selection in ISO Latin-1 format. See section 3.7.1 of the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**.
TEXT	TEXT	The text selection in the format preferred by the selection holder. See section 3.7.1 of the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**.
HOST_NAME	TEXT	The name of the machine running the client as seen from the machine running the server. See section 3.6.2 of the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**.
FILE_NAME	TEXT	The full path name of the files. See section 3.6.2 of the X/Open CAE Specification, **Window Management: File Formats and Application Conventions**.
_DT_NETFILE	TEXT	The full pathname of the files, each encoded using *tt_file_netfile*( ) and decoded using *tt_netfile_file*( ).
_DT_BUFFER_DATA	_DT_BUFFER_DATA	The buffer data in an array of bytes.
_DT_BUFFER_LENGTHS	INTEGER	The lengths in bytes of each buffer in the _DT_BUFFER_NAMES array.
_DT_BUFFER_NAMES	STRING	The names of each buffer, suitable for use as a file name.

Chapter 8

Data Typing

8.1 Introduction

The XCDE data typing services provide data capabilities that enhance the use of traditional file systems. These capabilities includes typing and attribute management.

The data typing services consist of the data criteria table, the data attributes table and the API used to access the tables. Data typing, using data criteria, can determine the data attributes of a file or byte vector, based on its name, file permissions, symbolic links and content. Data attributes determine user-visible interfaces to data: a human-readable description of the type, the icon to represent it graphically and the actions that apply to it. An object's data attributes also indicate the unique string that names the data interchange format to which the data's contents conform.

8.2 Functions

This section defines the functions, macros and external variables that provide XCDE data typing services to support application portability at the C-language source level.

NAME

DtDtsBufferToAttributeList — get a list of data attributes for a byte stream

SYNOPSIS

```
#include <Dt/Dts.h>

DtDtsAttribute **DtDtsBufferToAttributeList(const void *buffer,
                                            const int size,
                                            const char *opt_name);
```

DESCRIPTION

The *DtDtsBufferToAttributeList*( ) function returns a list of data attributes for a given byte stream.

The *buffer* argument is a pointer to the buffer of the data to be typed.

The *size* argument is the size of the buffer in bytes.

The *opt_name* argument can be used to specify a name to be associated with the buffer. If the *opt_name* argument is not NULL, it is used as a pseudo file name in typing; otherwise, certain attributes may be returned as NULL because the filename components could not be determined.

RETURN VALUE

Upon successful completion, the *DtDtsBufferToAttributeList*( ) function returns a NULL-terminated array of pointers of ***DtDtsAttribute**. If no value could be determined, it returns NULL.

APPLICATION USAGE

The application should use the *DtDtsFreeAttributeList*( ) function to release the memory for the returned value.

The *DtDtsBufferToAttributeList*( ) function assumes that the buffer is readable and writable by the user, group and other file classes and selects a type accordingly. An application requiring a type based on read-only permissions should use *DtDtsDataToDataType*( ).

SEE ALSO

<Dt/Dts.h>, *DtDtsDataToDataType*( ), *DtDtsLoadDataTypes*( ), *DtDtsFreeAttributeList*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsBufferToAttributeValue — get a single data attribute value for a byte stream

SYNOPSIS

```
#include <Dt/Dts.h>

char *DtDtsBufferToAttributeValue(const void *buffer,
                                  const int size,
                                  const char *attr_name,
                                  const char *opt_name);
```

DESCRIPTION

The *DtDtsBufferToAttributeValue*( ) function returns a data attribute value for a given byte stream. The *buffer* argument is a pointer to the buffer of the data to be typed.

The *size* argument is the size of the buffer in bytes.

The *attr_name* argument is a name of the attribute.

The *opt_name* argument can be used to specify a name to be associated with the buffer. If the *opt_name* argument is not NULL, it is used as a pseudo file name in typing; otherwise, certain attributes may be returned as NULL because the filename components could not be determined.

RETURN VALUE

Upon successful completion, the *DtDtsBufferToAttributeValue*( ) function returns a pointer to a data attribute value string, or NULL if no value could be determined.

APPLICATION USAGE

The application should use the *DtDtsFreeAttributeValue*( ) function to release the memory for the returned value.

The *DtDtsBufferToAttributeValue*( ) function assumes that the buffer is readable and writable by the user, group and other file classes and selects a type accordingly. An application requiring a type based on read-only permissions should use *DtDtsDataToDataType*( ).

SEE ALSO

<Dt/Dts.h>, *DtDtsDataToDataType*( ), *DtDtsLoadDataTypes*( ), *DtDtsFreeAttributeValue*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsBufferToDataType — get the data type for a byte stream

SYNOPSIS

```
#include <Dt/Dts.h>

char *DtDtsBufferToDataType(const void *buffer,
                            const int size,
                            const char *opt_name);
```

DESCRIPTION

The *DtDtsBufferToDataType*( ) function returns the data type name for a given byte stream.

The *buffer* argument is a pointer to the buffer of the data to be typed.

The *size* argument is the size of the buffer in bytes.

The *opt_name* argument can be used to specify a name to be associated with the buffer. If the *opt_name* argument is not NULL, it is used as a pseudo file name in typing; otherwise, certain attributes may be returned as NULL because the filename components could not be determined.

RETURN VALUE

Upon successful completion, the *DtDtsBufferToDataType*( ) function returns a pointer to a data type name string, or NULL if no value could be determined.

APPLICATION USAGE

The application should use the *DtDtsFreeDataType*( ) function to release the memory for the returned value.

The *DtDtsBufferToDataType*( ) function assumes that the buffer is readable and writable by the user, group and other file classes and selects a type accordingly. An application requiring a type based on read-only permissions should use *DtDtsDataToDataType*( ).

SEE ALSO

<Dt/Dts.h>, *DtDtsDataToDataType*( ), *DtDtsLoadDataTypes*( ), *DtDtsFreeDataType*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsDataToDataType — get the data type for a set of data

SYNOPSIS

```
#include <Dt/Dts.h>

char *DtDtsDataToDataType(const char *filepath,
                          const void *buffer,
                          const int size,
                          const struct stat *stat_buff,
                          const char *link_path,
                          const struct stat *link_stat_buff,
                          const char *opt_name);
```

DESCRIPTION

The *DtDtsDataToDataType*( ) function determines the data type of a set of data, based on the information given in the non-NULL pointer arguments to the function. The function gathers any additional information, if it is needed, to compensate for the NULL arguments. For example, if the *filepath* argument is given, but the *stat_buff* argument is NULL and a *stat_buff* value is required to determine the data type, *DtDtsDataToDataType*( ) calls the *stat*( ) function to obtain the information.

The *filepath* argument is the pathname of a file.

The *buffer* argument is a pointer to the buffer of the data to be typed.

The *size* argument is the size of the buffer in bytes.

The *stat_buff* argument is the buffer returned from a *stat*( ) or *fstat*( ) call for use in typing.

The *link_path* argument is the pathname of the target file pointed to by a symbolic link.

The *link_stat_buff* argument is the buffer returned from an *lstat*( ) call for use in typing.

The *opt_name* argument can be used to specify a name to be associated with the buffer. If the *opt_name* argument is not NULL, it is used as a pseudo file name in typing; otherwise, certain attributes may be returned as NULL because the filename components could not be determined.

RETURN VALUE

Upon successful completion, the *DtDtsDataToDataType*( ) function returns a pointer to a data type string, or NULL if no value could be determined.

APPLICATION USAGE

The *DtDtsDataToDataType*( ) function is generally used by applications such as file managers to improve performance. Typical applications should probably use *DtDtsFileToDataType*( ) or *DtDtsBufferToDataType*( ) instead.

The *DtDtsBuffer** functions assume that the buffer is readable and writable by the user, group and other file classes and select a type accordingly. An application requiring a type based on read-only permissions should use *DtDtsDataToDataType*( ) with a *stat_buff* value created for this purpose: all fields should NULL except for the *st_mode* field, which should be set to:

```
S_IFREG | S_IROTH | S_IRGRP | S_IRUSR
```

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ), *DtDtsFileToDataType*( ), *DtDtsBufferToDataType*( ); *fstat*( ), *lstat*( ), *stat*( ) in the X / Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**.

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsDataTypeIsAction — determine if the data type is an action

SYNOPSIS

```
#include <Dt/Dts.h>

int DtDtsDataTypeIsAction(const char *datatype);
```

DESCRIPTION

The *DtDtsDataTypeIsAction*( ) function determines if the specified data type is an action—a data type that was loaded from the action tables of the actions and data types database.

The *datatype* argument is a pointer to a data type name string.

RETURN VALUE

Upon successful completion, the *DtDtsDataTypeIsAction*( ) function returns non-zero if the data type is an action; otherwise, it returns zero.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsDataTypeNames — get a list of available data types

SYNOPSIS

```
#include <Dt/Dts.h>

char **DtDtsDataTypeNames(void);
```

DESCRIPTION

The *DtDtsDataTypeNames*( ) function returns a list of all available data types that are currently loaded into the data types database.

RETURN VALUE

Upon successful completion, the *DtDtsDataTypeNames*( ) function returns a NULL-terminated array of pointers to data type name strings.

APPLICATION USAGE

The application should use the *DtDtsFreeDataTypeNames*( ) function to release the memory for the returned value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ), *DtDtsFreeDataTypeNames*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsDataTypeToAttributeList — get a list of attributes for a data type

SYNOPSIS

```
#include <Dt/Dts.h>

DtDtsAttribute **DtDtsDataTypeToAttributeList(const char *datatype,
                                              const char *opt_name);
```

DESCRIPTION

The *DtDtsDataTypeToAttributeList*( ) function returns a list of attributes for a data type.

The *datatype* argument is a pointer to a data type name string.

The *opt_name* argument can be used to specify a name to be associated with the data type. If the *opt_name* argument is not NULL, it is used as a pseudo file name in typing; otherwise, certain attributes may be returned as NULL because the filename components could not be determined.

RETURN VALUE

Upon successful completion, the *DtDtsDataTypeToAttributeList*( ) function returns a NULL-terminated array of pointers of ∗**DtDtsAttribute**, or NULL if no value could be determined.

APPLICATION USAGE

The application should use the *DtDtsFreeAttributeList*( ) function to release the memory for the returned value.

The *opt_name* argument is useful when the attribute being returned contains a modifier string that depends on having a file name included. For example, if the INSTANCE_ICON attribute had the value **%name%.icon**, *opt_name* would be used to derive the **%name%** portion of the attribute value. See Section 8.4.4.16 on page 464.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ), *DtDtsFreeAttributeList*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsDataTypeToAttributeValue — get an attribute value for a specified data type

SYNOPSIS

```
#include <Dt/Dts.h>

char *DtDtsDataTypeToAttributeValue(const char *datatype,
                                    const char *attr_name,
                                    const char *opt_name);
```

DESCRIPTION

The *DtDtsDataTypeToAttributeValue*( ) returns an attribute value for the specified data type name.

The *datatype* argument is a pointer to a data type name string.

The *attr_name* argument is a name of the attribute.

The *opt_name* argument can be used to specify a name to be associated with the data type. If the *opt_name* argument is not NULL, it is used as a pseudo file name in typing; otherwise, certain attributes may be returned as NULL because the filename components could not be determined.

RETURN VALUE

Upon successful completion, the *DtDtsDataTypeToAttributeValue*( ) function returns a pointer to a data attribute value string, or NULL if no value could be determined.

APPLICATION USAGE

The application should use the *DtDtsFreeAttributeValue*( ) function to release the memory for the returned value.

The *opt_name* argument is useful when the attribute being returned contains a modifier string that depends on having a file name included. For example, if the INSTANCE_ICON attribute had the value **%name%.icon**, *opt_name* would be used to derive the **%name%** portion of the attribute value. See Section 8.4.4.16 on page 464.

EXAMPLES

The following takes a list of files as arguments and determines the description and actions for each file:

```
#include <Dt/Dts.h>

#define      ATTRIBUTE1   "DESCRIPTION"
#define      ATTRIBUTE2   "ACTIONS"

main (int argc, char **argv)
{
       char   *attribute;
       char   *datatype;

       /* load data types database */
       DtDtsLoadDataTypes();

       argv++;
       while (*argv) {
              /* get data type file file */
              datatype = DtDtsFileToDataType(*argv);

              /* get first attribute for datatype */
              attribute = DtDtsDataTypeToAttributeValue(datatype,
                                   ATTRIBUTE1, *argv);
              if (attribute)
```

```
                        printf("%s for file %s is %s\n",
                              ATTRIBUTE1, *argv, attribute);

                  /* get second attribute for datatype */
                  attribute = DtDtsDataTypeToAttributeValue(datatype,
                                    ATTRIBUTE2, NULL);
                  if (attribute)
                        printf("%s for file %s is %s\n",
                              ATTRIBUTE2, *argv, attribute);
                  argv++;
            }
            DtDtsRelease();
            exit(0);
      }
```

SEE ALSO

<Dt/Dts.h>, *DtDtsFileToDataType*( ), *DtDtsLoadDataTypes*( ), *DtDtsRelease*( ), *DtDtsFreeAttributeValue*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsFileToAttributeList — get a list of attributes for a file

SYNOPSIS

```
#include <Dt/Dts.h>

DtDtsAttribute **DtDtsFileToAttributeList(const char *filepath);
```

DESCRIPTION

The *DtDtsFileToAttributeList*( ) function returns a list of attributes for the specified file.

The *filepath* argument is the pathname of the file.

RETURN VALUE

Upon successful completion, the *DtDtsFileToAttributeList*( ) function returns a NULL-terminated array of pointers of ***DtDtsAttribute**, or NULL if no values could be determined.

APPLICATION USAGE

The application should use the *DtDtsFreeAttributeList*( ) function to release the memory for the returned value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ), *DtDtsFreeAttributeList*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsFileToAttributeValue — get a specified attribute value for a file

SYNOPSIS

```
#include <Dt/Dts.h>

char *DtDtsFileToAttributeValue(const char *filepath,
                                const char *attr_name);
```

DESCRIPTION

The *DtDtsFileToAttributeValue*( ) function returns a data attribute value for the specified file.

The *filepath* argument is the pathname of the file.

The *attr_name* argument is a pointer to an attribute name string.

RETURN VALUE

Upon successful completion, the *DtDtsFileToAttributeValue*( ) function returns a pointer to a data attribute value string, or NULL if no value could be determined.

APPLICATION USAGE

The application should use the *DtDtsFreeAttributeValue*( ) function to release the memory for the returned value.

EXAMPLES

The following takes a list of files as arguments and determines the description of the data type for each file:

```
#include <Dt/Dts.h>

#define      ATTRIBUTE    "DESCRIPTION"

main (int argc, char **argv)
{
       char   *attribute;

       /* load data types database */
       DtDtsLoadDataTypes();

       argv++;
       while (*argv) {
              /* get attribute for file */
              attribute = DtDtsFileToAttributeValue(*argv, ATTRIBUTE);

              if (attribute)
                     printf("%s: %s\n", *argv, attribute);
              argv++;
       }
       DtDtsRelease();
       exit(0);
}
```

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ), *DtDtsRelease*( ), *DtDtsFreeAttributeValue*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsFileToDataType — get a data type for a file

SYNOPSIS

```
#include <Dt/Dts.h>

char *DtDtsFileToDataType(const char *filepath);
```

DESCRIPTION

The function *DtDtsFileToDataType*( ) returns a data type name for the specified file.

The *filepath* argument is the pathname of the file.

RETURN VALUE

Upon successful completion, the *DtDtsFileToDataType*( ) function returns a pointer to a data type name string, or NULL if no value could be determined.

APPLICATION USAGE

The application should use the *DtDtsFreeDataType*( ) function to release the memory for the returned value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ), *DtDtsFreeDataType*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsFindAttribute — get a specified list of data types

SYNOPSIS

```
#include <Dt/Dts.h>

char **DtDtsFindAttribute(const char *attr_name,
                          const char *attr_value);
```

DESCRIPTION

The *DtDtsFindAttribute*( ) function returns the list of data types that have an attribute name that equals the specified value.

The *attr_name* argument is the attribute name.

The *attr_value* argument is the value of an attribute to be matched.

RETURN VALUE

Upon successful completion, the *DtDtsFindAttribute*( ) function returns a NULL-terminated array of data types, or NULL if no value could be determined.

APPLICATION USAGE

The application should use the *DtDtsFreeDataTypeNames*( ) function to release the memory for the returned value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ), *DtDtsFreeDataTypeNames*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsFreeAttributeList — free a list of data attributes

SYNOPSIS

```
#include <Dt/Dts.h>

void DtDtsFreeAttributeList(DtDtsAttribute **attr_list);
```

DESCRIPTION

The *DtDtsFreeAttributeList*( ) function frees the memory used for an attribute list.

The *attr_list* argument is a list of attribute and value pairs defined by the **DtDtsAttribute** structure.

RETURN VALUE

The *DtDtsFreeAttributeList*( ) function returns no value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsFreeAttributeValue — free a data attribute value

SYNOPSIS

```
#include <Dt/Dts.h>

void DtDtsFreeAttributeValue(char *attr_value);
```

DESCRIPTION

The *DtDtsFreeAttributeValue*( ) function frees the memory used for an attribute value.

The *attr_value* argument is the value of an attribute.

RETURN VALUE

The *DtDtsFreeAttributeValue*( ) function returns no value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsFreeDataType — free data type pointer memory

SYNOPSIS

```
#include <Dt/Dts.h>

void DtDtsFreeDataType(char *datatype);
```

DESCRIPTION

The *DtDtsFreeDataType*( ) function frees the memory used for a data type name.

The *datatype* argument is a pointer to a data type name string.

RETURN VALUE

The *DtDtsFreeDataType*( ) function returns no value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsFreeDataTypeNames — free a list of data type names

SYNOPSIS

```
#include <Dt/Dts.h>

void DtDtsFreeDataTypeNames(char **namelist);
```

DESCRIPTION

The *DtDtsFreeDataTypeNames*( ) function frees the memory used for a list of data type names.

The *namelist* argument is a list of data type names.

RETURN VALUE

The *DtDtsFreeDataTypeNames*( ) function returns no value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsIsTrue — return a Boolean value associated with a string

SYNOPSIS

```
#include <Dt/Dts.h>

Boolean DtDtsIsTrue(const char *string);
```

DESCRIPTION

The *DtDtsIsTrue*( ) function tests a string for a Boolean value. Any of the following *string* values, without regard to case, causes a return value of True:

```
true
yes
on
1
```

RETURN VALUE

The *DtDtsIsTrue*( ) function returns True if the *string* represents a true value; otherwise, it returns False.

SEE ALSO

<Dt/Dts.h>, *DtDtsBufferToAttributeList*( ), *DtDtsBufferToAttributeValue*( ), *DtDtsFileToAttributeList*( ), *DtDtsFileToAttributeValue*( ), *DtDtsDataTypeToAttributeList*( ), *DtDtsDataTypeToAttributeValue*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsLoadDataTypes — load and initialise the data types database

SYNOPSIS

```
#include <Dt/Dts.h>

void DtDtsLoadDataTypes(void);
```

DESCRIPTION

The *DtDtsLoadDataTypes*( ) function initialises and loads the database fields for the data typing functions.

APPLICATION USAGE

An alternative method to initialise and load the database is *DtDbLoad*( ).

RETURN VALUE

The *DtDtsLoadDataTypes*( ) function returns no value.

SEE ALSO

<Dt/Dts.h>, *DtDbLoad*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsRelease — free memory associated with the data types database

SYNOPSIS

```
#include <Dt/Dts.h>

void DtDtsRelease(void);
```

DESCRIPTION

The *DtDtsRelease*( ) function releases the data structures and data associated with the data types database, generally in preparation for a reload.

RETURN VALUE

The *DtDtsRelease*( ) function returns no value.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ), *DtDbLoad*( ).

CHANGE HISTORY

First released in Issue 1.

NAME

DtDtsSetDataType — set the data type of a directory

SYNOPSIS

```
#include <Dt/Dts.h>

char *DtDtsSetDataType(const char *dirpath,
                       const char *datatype,
                       const int override);
```

DESCRIPTION

The *DtDtsSetDataType*( ) function sets the data type of a directory. This may be accomplished by adding a file named with a leading dot to the directory.

The *dirpath* argument is a pathname of the directory.

The *datatype* argument is a data type.

If the value is already set, *DtDtsSetDataType*( ) does not change the value unless the *override* argument is set to True.

RETURN VALUE

Upon successful completion, the *DtDtsSetDataType*( ) function returns a pointer to a data type string, or NULL if it was unable to set or retrieve the data type.

APPLICATION USAGE

Directories can have data types associated with them, just as regular files can. For example, a file manager may choose to alter the appearance of the directory's icon based on this data type or a system may use a directory of files as an means of supporting a complex form of data, such as a compund document.

SEE ALSO

<Dt/Dts.h>, *DtDtsLoadDataTypes*( ).

CHANGE HISTORY

First released in Issue 1.

8.3 Headers

This section describes the contents of headers used by the XCDE data typing functions, macros and external variables.

Headers contain the definition of symbolic constants, common structures, preprocessor macros and defined types. Each function in Section 8.2 specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

NAME

Dt/Dts.h — data typing definitions

SYNOPSIS

```
#include <Dt/Dts.h>
```

DESCRIPTION

The **<Dt/Dts.h>** header declares the following structure:

```
typedef struct  _DtDtsAttribute {
       char    *name;
       char    *value;
} DtDtsAttribute;
```

The header declares the following as functions:

```
DtDtsAttribute **DtDtsBufferToAttributeList(const void *buffer,
                                            const int size,
                                            const char *opt_name);

char *DtDtsBufferToAttributeValue(const void *buffer,
                                  const int size,
                                  const char *attr_name,
                                  const char *opt_name);

char *DtDtsBufferToDataType(const void *buffer,
                            const int size,
                            const char *opt_name);

char *DtDtsDataToDataType(const char *filepath,
                          const void *buffer,
                          const int size,
                          const struct stat *stat_buff,
                          const char *link_path,
                          const struct stat *link_stat_buff,
                          const char *opt_name);

int DtDtsDataTypeIsAction(const char *datatype);

char **DtDtsDataTypeNames(void);

DtDtsAttribute **DtDtsDataTypeToAttributeList(const char *datatype,
                                              const char *opt_name);

char *DtDtsDataTypeToAttributeValue(const char *datatype,
                                    const char *attr_name,
                                    const char *opt_name);

DtDtsAttribute **DtDtsFileToAttributeList(const char *filepath);

char *DtDtsFileToAttributeValue(const char *filepath,
                                const char *attr_name);

char *DtDtsFileToDataType(const char *filepath);

char **DtDtsFindAttribute(const char *attr_name,
                          const char *attr_value);

void DtDtsFreeAttributeList(DtDtsAttribute **attr_list);
```

```
void DtDtsFreeAttributeValue(char *attr_value);

void DtDtsFreeDataType(char *datatype);

void DtDtsFreeDataTypeNames(char **namelist);

void DtDtsLoadDataTypes(void);

void DtDtsRelease(void);

char *DtDtsSetDataType(const char *dirpath,
                       const char *datatype,
                       const int override);
```

CHANGE HISTORY

First released in Issue 1.

8.4 Data Formats

8.4.1 Location of Actions and Data Types Database

The actions and data types database provides definitions for the actions and data types XCDE clients recognise. Files containing actions and data type definitions must end with the **.dt** suffix. The database is constructed by reading all files ending in the **.dt** suffix that are found in the search path specified by the *DTDATABASESEARCHPATH* environment variable.

The *DTDATABASESEARCHPATH* environment variable contains a comma-separated list of directories specified in **[***host***:]***/path* format. The *host*: portion is optional, but if specified, */path* is interpreted relative to *host*. In addition, *host* defines the *DatabaseHost* for records defined by files in the */path* directory. Otherwise, the *DatabaseHost* is the same as the *LocalHost*. To allow for localised action definitions, the data base search path supports the string **%L** within the pathname string. The logic that parses *DTDATABASESEARCHPATH* substitutes the value of the current locale as stored in the *LANG* environment variable for the string **%L** (or no characters if *LANG* is not set). Other uses of % within the *DTDATABASESEARCHPATH* pathnames produce unspecified results. Directories can be set up for various locales. Each directory contains localised action definitions for a single locale. For examples, see the default search path shown below. The local system administrator or the user (in **$HOME/.dtprofile**) can modify the actual value of the search path. The default search path includes the following directories, searched in the following sequence:

$HOME/.dt/types/
: personal user-defined database files

/etc/dt/appconfig/types/%L
: locally defined language-specific database files

/etc/dt/appconfig/types/C
: locally defined default database files

/usr/dt/appconfig/types/%L
: language-specific database files

/usr/dt/appconfig/types/C
: implementation-default database files

The XCDE data types database provides definitions for the data types and actions recognised by XCDE clients.

8.4.2 Data Types and Actions Database Syntax

The general syntax of the data types files is as follows:

```
set DtDbVersion=version_number
set VariableName=variable_value

RecordType record_name
{
       # Comment
       FieldName field_value
       FieldName field_value
              .
              .
              .
}
```

The set of general constructs composing the database entries is as follows:

8.4.2.1 Comments

Any line whose first non-space character is # is treated as a comment line, and is ignored during the reading of the database file.

8.4.2.2 Database Version

The database loader supports a version number, which indicates the version of the database syntax used by a particular database file. If a database version number is not specified, then the database loader assumes that the file uses the version 1.0 syntax, described here. If a database file specifies a version number, then it must be the first non-blank, non-comment line in the database file; if the version is specified anywhere else in the file, then an error message is generated, and the remainder of that database file is ignored. The database version number is specified using the following syntax:

```
set DtDbVersion=version_number
```

8.4.2.3 String Variables

Database records can reference string variables that are set within the database file. The scope of a string variable is restricted to only those record definitions within the database file defining the string variable. A string variable is defined using the following syntax:

```
set VariableName=variable_value
```

String variables are referenced using either of the standard shell variable referencing syntaxes: $*variable_name* or ${*variable_name*}. A variable name can be made up of any of the alphanumeric characters and the underscore. (See section 2.6.2 in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**.)

8.4.2.4 Environment Variables

Database records may refer to environment variables, using either of the standard shell variable referencing syntaxes: $*environment_variable* or ${*environment_variable*}. If the environment variable name conflicts with a string variable name, the string variable takes precedence.

8.4.2.5 Line Continuation

Any field within a record can be continued onto another line by ending the line with a \ character. The \ and any <blank>s following the \ and preceding the newline are discarded; leading <blank>s on the following line are preserved in the continued field.

8.4.2.6 Record Name

The first line of a record is made up of the record type, *RecordType* (one of: DATA_ATTRIBUTES, DATA_CRITERIA or ACTION), followed by the record name, *record_name*, which is henceforth used to identify this record. The *record_name* string must be coded in the codeset described in the referenced ISO/IEC 646:1983 standard and must be uniquely named across the data attributes, data criteria and actions tables.

8.4.2.7 Record Delimiters

After the record name has been located, the set of corresponding fields is delimited by the { and } characters. Each of these characters must appear on a line by itself.

8.4.2.8 Fields

The fields are all of the non-comment lines found between the record delimiters. They are composed of keyword/value pairs. The *FieldName* string must be coded in the codeset described in the referenced ISO/IEC 646:1983 standard. The *field_value* may be coded in additional, implementation-dependent, code sets, except that any literal string values shown in Section 8.4.3.11 on page 458 string must be coded in the codeset described in the referenced ISO/IEC 646:1983 standard.

8.4.2.9 Record Types

There are three recognised record types in database files used for data types (and actions):

- DATA_CRITERIA
- DATA_ATTRIBUTES
- ACTION

These three kinds of database record can appear together in the same file or they can be segregated into separate files. See Section 9.5 on page 489 for the file format of ACTION records.

8.4.3 Data Criteria Records

The first seven subsections of this section describe the *FieldName*s supported for data criteria records. The remaining subsections describe formatting and sorting information for data criteria records.

8.4.3.1 NAME_PATTERN Field

A shell pattern-matching expression describing the file names that could match this data. See section 2.13 of the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**. The default is an empty string, which means to ignore file patterns in matching.

If the data to be matched is in a buffer, rather than a file, the NAME_PATTERN expression is evaluated against the *opt_name* value given to *DtDtsBufferToDataType*( ) and related functions.

8.4.3.2 PATH_PATTERN Field

A shell pattern-matching expression describing the absolute pathnames that could match this data. See section 2.13 of the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**. The default is an empty string, which means to ignore path patterns in matching.

The PATH_PATTERN expression is used only for matching data in files; it does not affect matching of data in buffers.

8.4.3.3 CONTENT Field

Strings that match on the contents of a file, buffer or directory:

```
offset type value(s)
```

The *offset* string is a positive decimal integer number of octets from the beginning of the file or buffer, where the first *value* is tested. The *offset* value is ignored for the **filename** *type*.

The *type* string is one of the following:

string The *value* is a single string that is compared against the data starting at the *offset* location.

byte
short
long Each <blank>-separated *value* is an unsigned integer: decimal, octal (leading **0**) or hexadecimal (leading **0x** or **0X**). Multiple values are matched against multiple byte (octet), short (two octets) or long (four octets) locations starting at *offset* octets from the beginning of the file or data.

filename The *value* is a string that is compared against the filenames located anywhere in a directory. The use of **filename** on non-directory data produces undefined results.

The default CONTENT is an empty field, which means to ignore contents in matching.

The CONTENT field applies to data in both files and buffers.

Examples of two data criteria records with CONTENT fields are:

```
DATA_CRITERIA PCL1
{
       DATA_ATTRIBUTES_NAME    PCL
       CONTENT         0 byte 033 0105
       MODE            f&!x
}
DATA_CRITERIA POSTSCRIPT3
{
       DATA_ATTRIBUTES_NAME    POSTSCRIPT
       CONTENT         0 string %!
       MODE            f&!x
}
```

8.4.3.4 MODE Field

A string of zero to four characters that match the mode field of a *stat* structure (see **<sys/stat.h>** in the X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**). The first character indicates:

d match a directory

s match a socket

l match a symbolic link

f match a regular file

b match a block file

c match a character special file

The first, or subsequent characters, can also be:

r match any file with any of its user, group, or other read permission bits set

w match any file with any of its user, group, or other write permission bits set

x match any file with any of its user, group, or other execute or directory-search permission bits set

For example, the MODE field of **frw** matches any regular file that is readable or writable; **x** matches any file with any of its executable or search bits set.

The default is an empty field, which means to ignore the mode in matching.

If the data to be matched is in a buffer, rather than a file, the buffer is processed as if it had a mode of **fr**.

8.4.3.5 LINK_NAME Field

A shell pattern-matching expression describing the filename component (basename) of the filename the symbolic link points to that could match this data. See section 2.13 of the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**. The default is an empty expression, which means to ignore symbolic link names in matching. LINK_NAME points to the file itself, not to the name of the file.

The LINK_NAME expression is used only for matching data in files; it does not affect matching of data in buffers.

8.4.3.6 LINK_PATH Field

A shell pattern-matching expression describing the absolute pathname of the file pointed to by the symbolic link that could match this data. See section 2.13 of the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**. The default is an empty expression, which means to ignore symbolic link name in matching.

The LINK_PATH expression is used only for matching data in files; it does not affect matching of data in buffers.

8.4.3.7 DATA_ATTRIBUTES_NAME Field

The name of this type of data. This value is a *record_name* in the data attributes table.

8.4.3.8 Logical Expressions

The logical operators AND (&), OR (|) and NOT (!) can be used in any of the data criteria fields, except for DATA_ATTRIBUTES_NAME, as shown in Section 8.4.3.11 on page 458. The resultant expressions are evaluated from left to right.

8.4.3.9 White Space

White space is used to delimit tokens, as shown by the *blanks* and *newline* terminals in Section 8.4.3.11 on page 458. Within the *pattern* terminal, however, leading and trailing white space not explicitly shown in the grammar is significant to the expression. For example,

```
NAME_PATTERN   abc | def
```

is matched by either ‘‘**abc** ’’ (with a trailing <space>) or ‘‘ **def**’’ (with a leading <space>).

8.4.3.10 Escape Character

Shell pattern-matching and logical expression characters can be escaped and used as literal characters by preceding the character with a backslash (\). For example, * is interpreted as an asterisk, \? as a question mark and \[\] as square brackets. Backslash itself can be escaped by preceding it with a backslash (\ \).

8.4.3.11 Data Criteria Format

The following pseudo-BNF describes the data criteria variable definition:

```
DataCriteriaDefn   ::= 'DATA_CRITERIA' blanks record_name
{
                       data_criteria_defn
}

data_criteria_defn ::= (
                       'PATH_PATTERN' blanks pattern_datas newline
                     | 'NAME_PATTERN' blanks pattern_datas newline
                     | 'LINK_PATH' blanks pattern_datas newline
                     | 'LINK_NAME' blanks pattern_datas newline
                     | 'CONTENT' blanks content_fields newline
                     | 'MODE' blanks mode_specs newline
                     | 'DATA_ATTRIBUTES_NAME' blanks name newline
                       )
pattern_datas      ::= pattern_data [('&' | '|') pattern_datas]
pattern_data       ::= ['!'] pattern
pattern            ::= a shell pattern matching expression, as defined in section 2.13
                       of the X/Open CAE Specification, Commands and Utilities,
                       Issue 4, Version 2
mode_specs         ::= mode_spec [('&' | '|') mode_specs]
mode_spec          ::= (
                       type_spec [permission_spec]
                     | type_spec ('&' | '|') permission_spec
                       )
type_spec          ::= ['!'] type_char {type_char}
type_char          ::= ('d' | 'l' | 'f' | 's' | 'b' | 'c' )
permission_spec    ::= ['!'] permission_char {permission_char}
permission_char    ::= ('r' | 'w' | 'x')
content_fields     ::= content_field [('&' | '|') content_fields]
content_field      ::= (
                       ['!'] offset blanks 'string' blanks string
                     | ['!'] offset blanks 'byte'  blanks data_values
                     | ['!'] offset blanks 'short' blanks data_values
                     | ['!'] offset blanks 'long'  blanks data_values
                     | ['!'] offset blanks 'filename' blanks string
                       )
offset             ::= an unsigned decimal integer
data_values        ::= data_value [blanks data_values]
data_value         ::= an unsigned integer: decimal, octal (leading 0) or hexadecimal
                       (leading 0x or 0X)
name               ::= ( "A-Z" | "a-z" ) [name_char]
name_char          ::= { "A-Z" | "a-z" | "0-9" | "-" }
string             ::= a character string, not including <newline>
newline            ::= '\n'
blanks             ::= one or more <blank>s
```

8.4.3.12 Data Criteria Sorting

There may be multiple data criteria records that could match a file or data. This subsection describes the sorting process used by the XCDE data typing services. The more specific criteria are sorted toward the top of the list and the more general criteria toward the bottom. The data criteria record selected is the first match found on the resulting sorted list.

The following sorting rules are applied in sequence to each possible pair of data criteria records. If a rule determines that one data criteria record is more specific than another, the two records are positioned in the list so that the more specific comes before the less specific; otherwise, the next rule in sequence is applied.

1. Records are ordered by the fields specified within them:
    a. Records with both content and pattern fields (most specific)
    b. Records with only pattern fields
    c. Records with only content fields
    d. Records with neither content nor pattern fields (least specific)
2. Records are ordered based on the presence of any shell pattern-matching characters in their file name patterns (NAME_PATTERN or PATH_PATTERN):
    a. File names with no shell pattern-matching characters (most specific)
    b. File names with no shell pattern-matching characters in the final suffix (such as ∗**.c**)
    c. Others (least specific)
3. Records with a path pattern are more specific than records with a name pattern.
4. Records with a name pattern of ∗ are treated as if they have no name pattern.
5. Records are ordered based on the types of shell pattern-matching characters in their patterns:
    a. Patterns with any **?** (most specific)
    b. Patterns with any **[ ]**
    c. Patterns with any ∗ (least specific)
6. Records with path patterns that share leading pathname components are ordered as follows:
    a. The leading pathname components without shell pattern-matching characters are selected for comparison. (For example, **/foo/bar/bam/baz.?** and **/foo/bar/**∗**/baz** are evaluated as **/foo/bar/bam** and **/foo/bar** for this rule.)
    b. The selected paths are ordered so that the longest is more specific.
    c. If the selected paths are equal, the full path patterns are ordered based on the number and types of shell pattern-matching characters in their patterns, in the following sequence:
        i. Path patterns with fewer ∗ characters are more specific.
        ii. Path patterns with fewer **[ ]** characters are more specific.
        iii. Path patterns with fewer **?** characters are more specific.

d. If the path patterns are still of equal specificity, the one with the larger number of literal characters (those not used as shell pattern-matching special characters) in its pattern after the first non-literal character is more specific.

7. Records are ordered based on a character sorting of the path patterns, with the lowest value in collation sequence being more specific.

8. Records are ordered so that the one with more criteria is more specific. (For example, a record with a PATH_PATTERN, CONTENT and MODE is more specific than one with only a PATH_PATTERN.)

Two records still equal after executing the preceding rules are ordered in an unspecified sequence.

8.4.4 Data Attribute Records

The following *FieldName*s are supported for data attribute records. Each of the *FieldName*s is identical to the corresponding *name* member string of a *DtDtsAttribute* structure; see **<Dt/Dts.h>**.

8.4.4.1 DESCRIPTION Field

A textual description of the data that is suitable for presentation to a user requesting information about the data. The description should contain no formatting information such as tab or newline characters. The application that presents the information to the user formats the information. If this field is NULL or is not included in the data attribute record, the name of the data attribute should be used.

8.4.4.2 ICON Field

The name of the icon to use for this data. If this field is NULL or is not included in the data attribute record, a default value (**Dtactn** for an executable file or **Dtdata** for other data) should be used.

Icons are found by using the standard XCDE icon search path, so the value can be either an absolute pathname (for example, **/foo/icons/myicon.bm**), a relative pathname (for example, **icons/myicon.bm**) or a partial filename (for example, **myicon**). Partial filenames are preferred because they allow the XCDE icon search path to find the icon with the optimum size and depth for the current environment. See the **XCSA** specification, **Section 19.2**, **Icon Conventions**.

8.4.4.3 INSTANCE_ICON Field

The name of the icon to use for this instance of data. The contents of this field are as described in Section 8.4.4.2. If INSTANCE_ICON is set, the application should use it instead of ICON. If this field is NULL or is not included in the data attribute record, the ICON field should be used.

An example value of INSTANCE_ICON is **%name%.icon**, which would select an icon based on a specific filename, rather than on a generic data type. (See Section 8.4.4.16 on page 464.)

8.4.4.4 PROPERTIES Field

Keywords to indicate properties for this data. Valid values are **visible** and **invisible**. These provide guidance to an application such as a file manager about whether a file should be visibly displayed to the user.

If this field is NULL or is not included in the data attribute record, the visible property should be assumed.

8.4.4.5 ACTIONS Field

A comma-separated list of actions that can be performed on this data. This list refers to names in the action table for actions that can be performed on this data. If this field is NULL or is not included in the data attribute record, no action is available.

8.4.4.6 NAME_TEMPLATE Field

A string used to create a new file for data of this type. The string is intended to be passed to *sprintf*( ) with the file name as the single argument. For example: **%s.mif**. (See *sprintf*( ) in the X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**.) The default is empty. (This field is contrasted with the NAME_PATTERN field of the data criteria table in that the template is used to create a specific file, such as **%s.c**, whereas the pattern is used to find files, such as ∗**.c**).

8.4.4.7 IS_EXECUTABLE Field

A string-Boolean value that tells users of this data type that it can be executed as an application. If IS_EXECUTABLE is a true value (as determined by the *DtDtsIsTrue*( ) function), the data is executable; if this field is NULL, is not included in the data attribute record or is not true, then the data is considered not executable.

8.4.4.8 MOVE_TO_ACTION Field

The name of an action to be invoked when an object is moved to the current object using a drag and drop operation.

The MOVE_TO_ACTION, COPY_TO_ACTION and LINK_TO_ACTION fields cause an action to be invoked with the drop target as the first of the **DtActionArg** arguments to the *DtActionInvoke*( ) function, and the drag sources as the remaining **DtActionArg** arguments. However, if the drop target is an action itself, it is omitted from the **DtActionArg** list. For example, using the syntax of the *dtaction* utility, if objects A and B are dropped onto non-action object C:

```
dtaction action-name C A B
```

If C is an action:

```
dtaction action-name A B
```

8.4.4.9 COPY_TO_ACTION Field

The name of an action to be invoked when an object is copied to the current object using a drag and drop operation.

8.4.4.10 LINK_TO_ACTION Field

The name of an action to be invoked when an object is linked to the current object using a drag and drop operation.

8.4.4.11 IS_TEXT Field

A string-Boolean value that tells users of this data type that it is suitable for manipulation (viewing or editing) in a text editor or text widget. The IS_TEXT field should be set to a true value (as determined by the *DtDtsIsTrue*( ) function) if the data is textual in nature and if it should be presented to the user in textual form.

Criteria for making this determination include whether the data:

- consists of human language, or
- is generated and maintained manually, or
- is usefully viewable and editable in a text editor, or
- contains no (or only minimal) structuring and formatting information.

If the IS_TEXT field is a true value, this indicates that the data is eligible to be displayed directly by an application. That is, the application can load the data directly into a text editing widget (for example, XmText) instead of invoking an action on the data. An example of this occurs in the XCDE mail services: if the first part of a multipart message has IS_TEXT true, then it is displayed in the text area of the message view window. Otherwise, the text area will contain only message headers and the first part of the message will be displayed as an icon in the attachment pane. It is immaterial whether the data *can* be loaded into an XmText widget—even binary data can be—but rather whether the data *should* be loaded into an XmText widget.

Note that the IS_TEXT field differs from the **text** attribute of the MIME_TYPE field, which is the MIME content-type, as described in the referenced MIME RFCs. The MIME content-type determines whether the data consists of textual characters or byte values. If the data consists of textual characters and is labelled as **text/***, the IS_TEXT field determines whether it is appropriate for the data to be presented to users in textual form.

Examples of common data types include recommendations of the appropriate value of IS_TEXT:

- Human language encoded in ASCII:

  ```
  MIME_TYPE     text/plain
  IS_TEXT       true
  ```

  Note, however, that not everything that is ASCII should be presented directly to the user.

- Human language encoded in EUC, JIS, Unicode, or an ISO Latin charset:

  ```
  MIME_TYPE     text/plain; charset=XXX
  IS_TEXT       true
  ```

- CalendarAppointmentAttrs:

  ```
  MIME_TYPE     text/plain
  IS_TEXT       false
  ```

  Calendar appointments should be treated as opaque objects (editable only by the appointment editor) and not shown to the user as text.

- HTML (HyperText Markup Language):

  ```
  MIME_TYPE     text/html
  IS_TEXT       true
  ```

- PostScript:

  ```
  MIME_TYPE     application/postscript
  IS_TEXT       false
  ```

- C program source (C_SRC):

  ```
  MIME_TYPE     text/plain
  IS_TEXT       true
  ```

- Bitmaps and pixmaps (XBM and XPM):

```
MIME_TYPE     text/plain
IS_TEXT       false
```

- Project or module files for the XCDE application building services:

```
MIME_TYPE     text/plain
IS_TEXT       false
```

- Shell scripts:

```
MIME_TYPE     text/plain
IS_TEXT       false
```

- Encoded text produced by *uuencode* (see the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**):

```
MIME_TYPE     text/plain
IS_TEXT       false
```

- Manual pages:

```
MIME_TYPE     text/plain
IS_TEXT       false
```

8.4.4.12 MEDIA Field

The names in the MEDIA name space describe the form of the data itself. MEDIA names are used as ICCCM selection targets; they are named in the MEDIA field of data type records; and they are used in the *type* parameter of ToolTalk Media Exchange messages.

The MEDIA name space is a subset of the name space of selection target atoms as defined by the ICCCM; see X/Open CAE Specification, **Window Management: File Formats and Application Conventions**. All selection targets that specify a data format are valid MEDIA names, and all valid MEDIA names can be used directly as selection targets. Some selection targets specify an attribute of the selection (for example, LIST_LENGTH) or a side effect to occur (for example, DELETE), rather than a data format. These attribute selection targets are not part of the MEDIA name space.

8.4.4.13 MIME_TYPE Field

MEDIA is the XCDE internal, unique name for data types. However, other external naming authorities have also established name spaces. MIME (Multipurpose Internet Message Extensions), as described in the referenced MIME RFCs, is one of those external registries, and is the standard type name space for the XCDE mail system.

8.4.4.14 X400_TYPE Field

X.400 types are similar in structure to the MEDIA type, but are formatted using different rules and have different naming authorities.

8.4.4.15 DATA_HOST Attribute

The DATA_HOST attribute is not a field that can be added to the data attributes table when it is in a file, but it may be returned to an application reading attributes from the table. The data typing service adds this attribute automatically to indicate the host system from which the data type was loaded. If this field is NULL or is not included in the data attribute record, the data type was loaded from the local system.

8.4.4.16 Modifiers

The following modifiers can be used in the values of the data attributes to modify the runtime values:

%file% The full pathname of the file.

%dir% The directory component of the pathname. For example, for **/usr/src/file.c**, %dir% is **/usr/src**.

%name% The filename of the file. For example, for **/usr/src/file.c**, %name% is **file.c**.

%suffix% The suffix of the file. For example, for **/usr/src/file.c**, %suffix% is **c**.

%base% The basename of the file. For example, for **/usr/src/file.c**, %base% is **file**.

Strings enclosed in backquotes (‘) are processed with the *popen*( ) function (see X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**) and the output replaces the backquotes and string.

8.4.4.17 Data Attributes Format

The following pseudo-BNF describes the data attributes variable definition:

```
DataAttributesDefn   ::= 'DATA_ATTRIBUTES' blanks record_name
{
                         data_attributes_defn
}

data_attributes_defn ::= (
                         'DESCRIPTION' blanks string newline
                       | 'ICON' blanks string newline
                       | 'INSTANCE_ICON' blanks string newline
                       | 'PROPERTIES' blanks string {',' string} newline
                       | 'ACTIONS' blanks name {',' name} newline
                       | 'NAME_TEMPLATE' blanks string newline
                       | 'IS_EXECUTABLE' blanks string newline
                       | 'MOVE_TO_ACTION' blanks string newline
                       | 'COPY_TO_ACTION' blanks string newline
                       | 'LINK_TO_ACTION' blanks string newline
                       | 'IS_TEXT' blanks string newline
                       | 'MEDIA' blanks string newline
                       | 'MIME_TYPE' blanks string newline
                       | 'X400_TYPE' blanks string newline
                       | unique_string blanks string newline
                       | '#' string newline
                         )
string               ::= a character string, not including <newline>
newline              ::= '\n'
unique_string        ::= a uniquely named string for implementation extensions
blanks               ::= one or more <blank>s
```

8.4.4.18 Examples

The following are examples of data attribute and data criteria entries in the data typing database:

```
DATA_ATTRIBUTES C_SRC
{
        ACTIONS         Open,Make,Print
        ICON            DtdotC
        IS_TEXT         true
        NAME_TEMPLATE   %s.c
        DESCRIPTION     A C_SRC file is a source file in the C \
                        programming language.
}

DATA_CRITERIA C_SRC1
{
        DATA_ATTRIBUTES_NAME C_SRC
        MODE            f
        NAME_PATTERN    *.c
}

DATA_ATTRIBUTES POSTSCRIPT
{
        ACTIONS         Open,Print
        ICON            Dtps
        NAME_TEMPLATE   %s.ps
        MEDIA           POSTSCRIPT
        MIME_TYPE       application/postscript
}

DATA_CRITERIA POSTSCRIPT1
{
        DATA_ATTRIBUTES_NAME POSTSCRIPT
        MODE            fr
        NAME_PATTERN    *.ps
}
```

Chapter 9

Execution Management

9.1 Introduction

The XCDE execution management service is the infrastructure and API that can send a message to, or invoke a process on, the appropriate system with the necessary supporting environment (for example, running inside a terminal emulator).

9.1.1 Scope

The execution management service allows existing command-line or message-based applications and utilities to be encapsulated in an object-oriented manner such that they can be accessed from the desktop or other software.

9.1.2 Components

The major components of the XCDE execution management service are:

action database
The database that defines actions that encapsulate applications and utilities. This database can be distributed across multiple systems and customised on a network, system or personal scope.

action invocation library
The client-side library that provides an API for loading the database, querying the database and invoking actions.

dtexec client
The utility that controls processes spawned as the result of invoking an action and reports changes in their status. The interface to the *dtexec* client is implementation-specific.

9.1.3 Action Database Entries

Entries in the action database provide the following information:

icon The icons that represent the action.

arguments
The number and kind of arguments that the action accepts.

description
A textual description of the action.

type Whether the action encapsulates a command line or a message.

Message-based actions provide additional information that specifies the type of message to be sent and the details of the message.

9.1.4 Command-Line Actions

Command-line actions provide some or all of the following additional information:

window type
The type of window support (for example, none or a terminal emulator) required by the command.

directory
The current working directory where the command must execute.

execution host
The host (or list of possible hosts) where the command must execute.

command
The command line that is to be executed.

9.2 Functions

This section defines the functions, macros and external variables that provide XCDE execution management services to support application portability at the C-language source level.

NAME

DtActionCallbackProc — notify application that the status of an application has changed

SYNOPSIS

```
#include <Dt/Action.h>
```

DESCRIPTION

The **<Dt/Action.h>** header defines the *DtActionCallbackProc*( ) callback prototype as follows:

```
typedef void (*DtActionCallbackProc)(DtActionInvocationID id,
                                     XtPointer client_data,
                                     DtActionArg *args,
                                     int argCount,
                                     DtActionStatus status);
```

If registered when invoking an action with *DtActionInvoke*( ), a *DtActionCallbackProc*( ) procedure is called whenever an action has a status update, such as action termination. Depending on *status*, modified action arguments may be returned using *args*.

The *id* argument specifies an invocation ID as returned by *DtActionInvoke*( ).

The *client_data* argument specifies the client data that was registered with *DtActionInvoke*( ).

The *args* argument is an array of updated action argument structures, if there are any. Individual arguments have their *argClass* set to one of the standard argument classes, or DtACTION_NULLARG, to indicate that the current status update is not providing an update for the given argument. If the object has been removed (for example, dropped on the trash), the return *argClass* is set to DtACTION_NULLARG to indicate that it no longer exists.

The *args* array has been allocated by *XtMalloc*( ), as have any of the **char*** or **void*** elements contained in each of the *args*. The application is responsible for calling *XtFree*( ) on all elements contained in each of the *args*, and then calling *XtFree*( ) on *args*.

The *argCount* argument specifies the total number of arguments in *args*. This number equals the number of arguments originally provided to *DtActionInvoke*( ).

The *n*th argument in the original action argument array corresponds to the *n*th argument in an updated action argument array.

The *status* argument specifies the purpose of the status update. The status codes listed here and in **<Dt/Action.h>**, may be returned in a *DtActionCallbackProc*( ):

DtACTION_INVOKED
: The corresponding *DtActionInvoke*( ), which is asynchronous and does not block when starting actions, has finished starting the requested actions. For all *DtActionInvoke*( ) calls that include a *DtactionCallbackProc*( ), this status code is guaranteed to be returned. When returned, no additional prompts for data will appear from the action service.

DtACTION_DONE
: The actions that were the result of the original *DtActionInvoke*( ) call have terminated normally. Once this status value is returned, all registered callbacks are invalidated, and *id* can no longer be used in subsequent action service calls. Returned *args* data may accompany the DtACTION_DONE status code. For all *DtActionInvoke*( ) calls that include a *DtActionCallbackProc*( ), this status code or an equivalent status code (for example, DtACTION_CANCELED or DtACTION_FAILED) is guaranteed to be returned.

DtACTION_CANCELED
: The actions that were the result of the original *DtActionInvoke*( ) call were canceled and have terminated normally. Once this status value is returned, all registered callbacks are invalidated, and *id* can no longer be used in subsequent action service calls. No *args* data will accompany the DtACTION_CANCELED status code.

DtACTION_FAILED
: An error occured and a normal termination is no longer possible. The action service may have failed to start the action or lost contact with and abandoned the action. Once this status value is returned, an error dialog may be posted by the action service, all registered callbacks are invalidated, and *id* can no longer be used in subsequent action service calls. No *args* data will accompany the DtACTION_FAILED status code.

DtACTION_STATUS_UPDATE
: The actions associated with *id* have generated a status update, such as returning modified *args*. Updates occur in several ways.

  If several actions were started from a single *DtActionInvoke*( ), then as each individual action terminates, a DtACTION_STATUS_UPDATE with return *args* is returned, and when the final action terminates, a DtACTION_DONE or equivalent status code is returned, possibly with return arguments.

  Other actions may have the capability to generate updates (for example, Tooltalk-based actions doing a Media Exchange Deposit (Request)).

  In most cases, a *DtActionArg* argument array accompanying a DtACTION_STATUS_UPDATE only has updated data for a few of the arguments; the remaining arguments are set to DtACTION_NULLARG.

EXAMPLES

The following shows how a *DtActionCallbackProc*( ) might be coded.

```
DtActionCallbackProc myCallback(
    DtActionInvocationID id,
    XtPointer client_data,
    DtActionArg *actionArgPtr,
    int actionArgCount,
    DtActionStatus status);
{
    extern DtActionArg *myUpdatedArgs; /* global hook for new data */
    extern int myDoneFlag; /* global done flag */

    switch (status) {
        case DtACTION_INVOKED:
            /*
             * All the arguments to the original DtActionInvoke
             * have been consumed by actions, and the actions have
             * been started.  Among other things, we will not see
             * any more prompts for user input.
             */
            break;
        case DtACTION_DONE:
            myUpdatedArgs = (DtActionArg *) actionArgPtr;
            myDoneFlag = TRUE;
            break;
```

```
            case DtACTION_CANCELED:
            case DtACTION_FAILED:
                if ((actionArgCount != 0) && actionArgPtr) {
                    /*
                     * If not a normal shutdown, throw away returned
                     * information.
                     */
                    for (i=0; i < actionArgCount; i++) {
                        if (actionArgPtr[i].argClass == DtACTION_FILE) {
                            XtFree(actionArgPtr[i].u.file.name);
                        } else if (actionArgPtr[i].argClass ==
                                    DtACTION_BUFFER) {
                            XtFree(actionArgPtr[i].u.buffer.bp);
                            XtFree(actionArgPtr[i].u.buffer.type);
                            XtFree(actionArgPtr[i].u.buffer.name);
                        }
                    }
                    XtFree(actionArgPtr);
                }
                myUpdatedArgs = (DtActionArg *) NULL;
                myDoneFlag = FALSE;
                break;
            case DtACTION_STATUS_UPDATE:
                myUpdatedArgs = (DtActionArg *) actionArgPtr;
                myDoneFlag = FALSE;
                break;
            default:
                /* ignore */
                break;
        }
    }
```

SEE ALSO

<Dt/Action.h>, *DtDbLoad*( ), *DtActionLabel*( ), *DtActionDescription*( ), *DtActionExists*( ), *DtActionInvoke*( ), *DtActionType*( ); *XtMalloc*( ), *XtFree*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**; Section 9.5 on page 489.

CHANGE HISTORY

First released in Issue 1.

NAME

DtActionDescription — obtain the descriptive text for a given action

SYNOPSIS

```
#include <Dt/Action.h>

char *DtActionDescription(char *actionName);
```

DESCRIPTION

The *DtActionDescription*( ) function looks up and returns the descriptive text associated with the *actionName* action. The *actionName* argument is the name of the action. If there are multiple *actionName* actions, the string returned is the description of the most general. The most general action is the one with the lowest precedence, as described in Section 9.5.2.7 on page 491.

RETURN VALUE

Upon successful completion, the *DtActionDescription*( ) function returns a newly allocated copy of the description string associated with the action; otherwise, it returns NULL.

APPLICATION USAGE

The *DtActionDescription*( ) function is useful for applications that wish to present information to the user about a particular action.

The application should use *XtFree*( ) to free the description string returned by *DtActionDescription*( ).

SEE ALSO

<Dt/Action.h>, *XtFree*( ) in the X / Open CAE Specification, **Window Management**: **X Toolkit Intrinsics**; Section 9.5 on page 489, Section 8.4.1 on page 453.

CHANGE HISTORY

First released in Issue 1.

NAME

DtActionExists — determine if a string corresponds to an action name

SYNOPSIS

```
#include <Dt/Action.h>

Boolean DtActionExists(char *name);
```

DESCRIPTION

The *DtActionExists*( ) function checks whether a given name corresponds to an action name. The *name* argument is the name of the action.

RETURN VALUE

Upon successful completion, the *DtActionExists*( ) function returns True if *name* corresponds to an existing action name; otherwise, it returns False.

APPLICATION USAGE

The *DtActionExists*( ) function is useful for applications that need to verify that an action name is valid before attempting to invoke it.

SEE ALSO

<Dt/Action.h>, Section 9.5 on page 489, Section 8.4.1 on page 453.

CHANGE HISTORY

First released in Issue 1.

NAME

DtActionIcon — get the icon information for an action

SYNOPSIS

```
#include <Dt/Action.h>

char *DtActionIcon(char *actionName);
```

DESCRIPTION

The *DtActionIcon*( ) function gets the icon information for an action. The *actionName* argument is the name of the action. *DtActionIcon*( ) returns the name of the icon associated with an *actionName* action. If the action definition does not explicitly identify an icon name, this function returns the default action icon name, as described in Section 9.5.2.7 on page 491. The default action icon name can be customised using the **actionIcon** X resource.

If there are multiple *actionName* actions, the string returned is the icon associated with the most general action. The most general action is the one with the lowest precedence, as described in Section 9.5.2.7 on page 491.

RETURN VALUE

Upon successful completion, the *DtActionIcon*( ) function returns a newly allocated copy of the icon name string (ICON field) associated with the action; otherwise, it returns NULL.

APPLICATION USAGE

The *DtActionIcon*( ) function is useful for applications that provide a graphical interface to actions.

The application should use *XtFree*( ) to free the icon name string returned by the *DtActionIcon*( ) function.

SEE ALSO

<Dt/Action.h>, *XtFree*( ) in the X / Open CAE Specification, **Window Management**: **X Toolkit Intrinsics**; Section 9.5 on page 489, Section 8.4.1 on page 453.

CHANGE HISTORY

First released in Issue 1.

NAME

DtActionInvoke — invoke an XCDE action

SYNOPSIS

```
#include <Dt/Action.h>

DtActionInvocationID
DtActionInvoke(Widget w,
               char *action,
               DtActionArg *args,
               int argCount,
               char *termOpts,
               char *execHost,
               char *contextDir,
               int useIndicator,
               DtActionCallbackProc statusUpdateCb,
               XtPointer client_data);
```

DESCRIPTION

The *DtActionInvoke*( ) function provides a way for applications to invoke desktop actions on file or buffer arguments. Applications can register a callback for receiving action-done status and return arguments.

The actions and data types databases must be initialised and loaded (using *DtInitialize*( ) and *DtDbLoad*( )) before *DtActionInvoke*( ) can run successfully.

The *w* argument is a widget that becomes the parent of any dialogs or error messages resulting from action invocation. This widget should be a top-level application shell widget that continues to exist for the action's expected lifetime. This argument must have a non-NULL value.

The *action* argument is the name of the action to be invoked. The action database may define more than one action with the same name. The action selected for a particular invocation depends on the class, type, and number of arguments provided (as described in Section 9.5 on page 489). This argument must have a non-NULL value.

The *args* argument is an array of action argument structures containing information about the arguments for this action invocation. If there are no arguments, the value of *args* must be NULL. The items in this array are assigned to the option argument keywords referenced in the action definition (see Section 9.5 on page 489). The *n*th item is assigned to keyword %Arg_*n*%. For example, the second item is assigned to %Arg_*2*%.

The *argCount* argument is the number of action arguments provided in the array *args* references.

The *termOpts* argument is a string providing special execution information for the terminal emulator used for COMMAND actions of WINDOW_TYPE TERMINAL or PERM_TERMINAL. (See Section 9.5 on page 489). This string must be quoted if it contains embedded blanks. The application uses this string to pass on title, geometry, colour and font information to the terminal emulator. This information must be in a form the expected terminal emulator recognises. This argument can be NULL.

The *execHost* argument is a string identifying a preferred execution host for this action. The *execHost* specified here, supersedes the list of execution hosts defined in the action definition. If *execHost* is NULL, the execution host for the action is obtained from the action definition as described in Section 9.5 on page 489.

The *contextDir* argument is a string identifying a fallback working directory for the action. File name arguments are interpreted relative to this directory, which must reside in the local file

name space (for example, **/usr/tmp** or **/net/hostb/tmp**). This value is only used if the action definition does not explicitly specify a working directory in the CWD field of the action definition. If *contextDir* is NULL, the current working directory of the action is obtained from the action definition, as described in Section 9.5 on page 489.

If the *useIndicator* flag is zero, *DtActionInvoke*( ) does not provide any direct indication to the user that an action has been invoked. If the *useIndicator* flag is non-zero, the user is notified via some activity indicator (for example, a flashing light in the front panel) that an action has been invoked. This indication persists only until the invocation of the action completes (in other words, until the action begins running).

The *statusUpdateCb* callback may be activated if the invoked actions have returnable status (for example, a TT_MSG(TT_REQUEST) returning DtACTION_DONE). At a minimum, a DtACTION_INVOKED status is returned when *DtActionInvoked*( ) has finished processing and has completely invoked any resulting actions, and a DtACTION_DONE or equivalent done status is returned when all actions terminate. If *statusUpdateCb* is set to NULL, subsequent action status is not returned. (See **<DtAction.h>** for a list of all **DtActionStatus** codes, and see *DtActionCallbackProc*( ) for details on *statusUpdateCb* and a list of specific **DtActionStatus** codes it can return.)

The *client_data* argument is optional data to be passed to the *statusUpdateCb* callback when invoked.

The *DtActionInvoke*( ) function searches the action database for an entry that matches the specified action name, and accepts arguments of the class, type and count provided.

If *DtActionInvoke*( ) finds a matching action, the supplied arguments are inserted into the indicated action fields. If any missing action arguments have an associated prompt string, then a dialog box prompts the user to supply the arguments; otherwise, missing arguments are ignored. If too many arguments are supplied to an action requiring more than a single argument, a warning dialog is posted, allowing the action to be cancelled or continued, ignoring the extra arguments. If too many arguments are supplied to an action requiring zero or one arguments, then that action is invoked once for each of the supplied arguments. Arguments in the **DtActionArg** structure that may have been modified by the action are returned by the callback if a *statusUpdateCb* callback is provided. For **DtActionBuffer** arguments, the writable flag acts as a hint that the buffer is allowed to be modified and returned.

The **DtActionBuffer** structure contains at least the following members:

`void`	`*bp`	location of buffer
`int`	`size`	size of buffer in bytes
`char`	`*type`	optional type of buffer
`char`	`*name`	optional name of buffer
`Boolean`	`writable`	action is allowed to modify and return the buffer

The **DtActionFile** structure contains at least the following member:

`char`	`*name`	name of file

The **DtActionArg** structure contains at least the following members:

`int`	`argClass`	see argument class types (ARG_CLASS field)
`DtActionFile`	`u.file`	union to a DtActionFile structure
`DtActionBuffer`	`u.buffer`	union to a DtActionBuffer structure

where *argClass* is DtACTION_FILE or DtACTION_BUFFER. The action service may set *argClass* to DtACTION_NULLARG for action arguments returned by a *statusUpdateCb* to indicate that the argument is not being updated or has been removed. DtACTION_NULLARG cannot be present in action arguments passed to *DtActionInvoke*( ).

The *DtActionInvoke*( ) function accepts a pointer to an array of **DtActionArg** structures describing the objects to be provided as arguments to the action. The *args* structure can be modified or freed after *DtActionInvoke*( ) returns.

A single call to *DtActionInvoke*( ) may initiate several actions or messages. For example, if an action is given three files, but only needs one, three instances of the action are started, one for each file. As a result, a single returned **DtActionInvocationID** may represent a group of running actions, and subsequent execution management services calls operate on that group of actions.

For DtACTION_BUFFER arguments, the action service first tries to type the buffer ∗*bp* using the *name* field (see Section 8.4 on page 453). The *name* field would typically contain a value resembling a file name with an optional extension describing its type. If the *name* field is NULL, then the action service uses the type specified in the *type* field. If the *type* field is NULL, then the action service types the buffer ∗*bp* by content (see Section 8.4 on page 453). If the *name* and *type* fields are both non-NULL, then the action service uses the *name* field for typing and ignores the *type* field. If the buffer pointer *bp* is NULL or *size* is equal to zero, a buffer with no contents is used in the resulting action. If returned, the buffer pointer *bp* is defined, and *size* is equal to or greater than zero.

When necessary, DtACTION_BUFFER arguments are automatically converted to temporary files prior to actual action invocation, and reconverted back to buffers after action termination (this is transparent to the caller). If a non-NULL *name* field is given, it is used in the construction of the temporary file name (for example, **/myhome/.dt/tmp/name**). If the use of *name* would cause a conflict with an existing file, or *name* is NULL, the action service generates a temporary file name. The permission bits on the temporary file are set according to the *writable* field and the IS_EXECUTABLE attribute from the action service associated with the *type* field.

For DtACTION_FILE arguments, *name* is required.

For DtACTION_BUFFER arguments, *name* cannot contain slash characters.

Errors encountered are either displayed to the user in a dialog box or reported in the desktop errorlog file (**$HOME/.dt/errorlog**, unless configured otherwise).

RESOURCES

This section describes the X11 resources the *DtActionInvoke*( ) function recognises. The resource class string always begins with an upper-case letter. The corresponding resource name string begins with the lower case of the same letter. These resources can be defined for all clients using the Action Library API by specifying ∗**resourceName: value**.

X11 Resources			
Name	**Class**	**Value Type**	**Default**
localterminal	**LocalTerminal**	**string**	‘‘Dtterm’’
remoteTerminals	**RemoteTerminals**	**string**	‘‘Dtterm’’
waitTime	**WaitTime**	**number**	3

LocalTerminal

Defines an alternative local terminal emulator for Command actions of WINDOW_TYPE TERMINAL or PERM_TERMINAL to use. The default terminal emulator is *dtterm*.

RemoteTerminals

Defines a comma-separated list of host and terminal emulator pairs. When a remote COMMAND action is executed on one of the hosts in the list, the terminal emulator associated with that host is used for that command. The list is passed to the terminal emulator using the **–e** argument. (Thus, if another terminal emulator than *dtterm* is used, it must support the **–e** argument.

WaitTime

Used to assign an alternative integer value, in seconds, to the threshold successful return time interval. If a COMMAND action of WINDOW_TYPE TERMINAL fails, the terminal emulator may be unmapped before the user has a chance to read the standard error from the failed command. This resource provides a workaround to this problem. If a TERMINAL window command exits before **WaitTime** seconds have elapsed, the terminal emulator window is forced to remain open, as if it were of TYPE PERM_TERMINAL. The default value of **WaitTime** is 3 seconds.

RETURN VALUE

Upon successful completion, the *DtActionInvoke*( ) function returns a **DtActionInvocationID**. The ID can be used in subsequent execution management services calls to manipulate the actions while they are running. **DtActionInvocationID**s are only recycled after many have been generated.

APPLICATION USAGE

The caller should allocate space for the array of structures describing the objects to be provided as arguments to the action. The caller can free the memory after *DtActionInvoke*( ) returns.

Since *DtActionInvoke*( ) spawns subprocesses to start local actions, the caller should use *waitpid*( ) instead of *wait*( ) to distinguish between processes started by the action service and those the caller starts.

EXAMPLES

Given the following action definition:

```
ACTION Edit
{
        LABEL           "Text Edit Action"
        ARG_CLASS       BUFFER, FILE
        ARG_TYPE        TEXT
        TYPE            COMMAND
        WINDOW_TYPE     TERMINAL
        EXEC_STRING     "textedit %Args%"
        DESCRIPTION     This action invokes the "textedit" command on
                        an arbitrary number of arguments.  A terminal
                        emulator is provided for this action's I/O.
                        EXEC_HOST and CWD are not specified so the
                        defaults are used for both quantities.
}
```

The following call invokes the action **Edit** on the arguments *aap* supplies:

```
DtActionInvoke(w, "Edit", aap, 3, NULL, NULL, NULL, 1,
                myCallback, myClientData);
```

The working directory for the action defaults to the current working directory. The execution host is the default execution host.

If the variable *aap* points to an array of **ActionArg** data structures containing the following information:

```
{
        argClass = DtACTION_FILE;
        struct {
            name="/myhome/file1.txt";
        } file;
}
{
        argClass = DtACTION_FILE;
        struct {
            name="file2.txt";
        } file;
}
{
        argClass = DtACTION_BUFFER;
        struct {
            bp=(void *) myEditBuffer;
            size=lengthOfMyEditBuffer;
            type=NULL;
            name="Doc1.txt"
            writable=TRUE;
        } buffer;
}
```

and the current working directory is **/cwd**, then the *Edit* action results in the execution string:

```
textedit /myhome/file1.txt /cwd/file2.txt /myhome/.dt/tmp/Doc1.txt
```

When the action completes, *myCallback* is called and the callback returns the buffer argument.

SEE ALSO

<Dt/Action.h>, *XtFree*( ), *XtMalloc*( ) in the X/Open CAE Specification, **Window Management: X Toolkit Intrinsics**; *DtDbLoad*( ), *DtInitialize*( ), *DtActionCallbackProc*( ), Section 9.5 on page 489, Section 8.4.1 on page 453, Section 8.4 on page 453.

CHANGE HISTORY

First released in Issue 1.

NAME

DtActionLabel — get the localizable label string for an action

SYNOPSIS

```
#include <Dt/Action.h>

char *DtActionLabel(char *actionName);
```

DESCRIPTION

The *DtActionLabel*( ) function provides access to the localizable label string associated with an action named *actionName*. The *actionName* argument is the name of the action. The localizable label string is the string that all components should display to identify the action. If the action definition does not specify a label string, the action name itself is returned.

The label string associated with an action is localizable, but the action name is not.

If there are multiple *actionName* actions, the label string returned is the label associated with the most general action. The most general action is the one with the lowest precedence, as described in Section 9.5.2.7 on page 491.

RETURN VALUE

Upon successful completion, the *DtActionLabel*( ) function returns a newly allocated copy of the localizable label string associated with the action if an action named *actionName* is found; otherwise, it returns NULL. It is up to the caller to free the memory associated with this new copy of the label. The default value for an action label is the action name itself.

APPLICATION USAGE

All applications displaying action names should use the action label to identify an action.

SEE ALSO

<Dt/Action.h>, Section 9.5 on page 489, Section 8.4.1 on page 453.

CHANGE HISTORY

First released in Issue 1.

NAME

DtDbLoad — load actions and data types database

SYNOPSIS

```
#include <Dt/Action.h>

void DtDbLoad(void);
```

DESCRIPTION

The *DtDbLoad*( ) function loads the actions and data types database into the application. When the function returns, the database has been loaded. See Section 8.4.1 on page 453 for the general syntax and location of the actions and data types database.)

RETURN VALUE

The *DtDbLoad*( ) function returns no value.

APPLICATION USAGE

If this function is used in a long-lived application, the application must dynamically reload the databases when they are modified. To do this, the client must register to receive notification whenever the actions and data types database needs to be modified. It is up to the application to recall *DtDbLoad*( ) after receiving notification. This is done with a call to *DtDbReloadNotify*( ).

If errors are encountered when reading the database files, error messages are written to the user's errorlog file (**$HOME/.dt/errorlog**). Records containing errors are not incorporated into the internal database.

SEE ALSO

<Dt/Action.h>, *DtDbReloadNotify*( ), Section 8.4.1 on page 453.

CHANGE HISTORY

First released in Issue 1.

NAME

DtDbReloadNotify — reload the Dt actions and data typing services database

SYNOPSIS

```
#include <Dt/Action.h>

void DtDbReloadNotify(DtDbReloadCallbackProc callback_proc,
                      XtPointer client_data);
```

DESCRIPTION

The *DtDbReloadNotify*( ) function registers an application callback function that is called whenever the actions and data types database needs to be reloaded; the conditions that trigger this callback are implementation-dependent. The *callback_proc* must flush any actions and data type information that the application has cached and then call *DtDbLoad*( ) to reload the database. The *client_data* argument passes additional application information to the callback routine.

RETURN VALUE

The *DtDbReloadNotify*( ) function returns no value.

If errors are encountered when reading the database files, error messages are written to the user's errorlog file (**$HOME/.dt/errorlog**). Records containing errors are not incorporated into the internal database.

SEE ALSO

<Dt/Action.h>, *DtDbLoad*( ), Section 9.5 on page 489, Section 8.4.1 on page 453.

CHANGE HISTORY

First released in Issue 1.

9.3 Headers

This section describes the contents of headers used by the XCDE execution management functions, macros and external variables.

Headers contain the definition of symbolic constants, common structures, preprocessor macros and defined types. Each function in Section 9.2 specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

NAME

Dt/Action.h — action service definitions

SYNOPSIS

```
#include <Dt/Action.h>
```

DESCRIPTION

The **<Dt/Action.h>** header defines the following DtActionStatus constants:

```
DtACTION_OK
DtACTION_INVALID_ID
DtACTION_INVOKED
DtACTION_DONE
DtACTION_CANCELED
DtACTION_FAILED
DtACTION_STATUS_UPDATE
```

The header defines the following data type through **typedef**:

```
typedef unsigned long DtActionInvocationID;
```

The header defines the following callback prototypes:

```
typedef void (*DtActionCallbackProc)(DtActionInvocationID id,
                                     XtPointer client_data,
                                     DtActionArg *args,
                                     int argCount,
                                     DtActionStatus status);

typedef void (*DtDbReloadCallbackProc)(XtPointer clientData);
```

The header defines the following as functions:

```
void DtDbReloadNotify(DtDbReloadCallbackProc proc,
                      XtPointer clientData);

void DtDbLoad(void);

Boolean DtActionExists(char *s);

char *DtActionLabel(char *s);

char *DtActionDescription(char *actionName);

char *DtActionIcon(char *actionName);

DtActionInvocationID
DtActionInvoke(Widget w,
               char *action,
               DtActionArg *args,
               int argCount,
               char *termOpts,
               char *execHost,
               char *contextDir,
               int useIndicator,
               DtActionCallbackProc statusUpdateCb,
               XtPointer client_data);
```

CHANGE HISTORY

First released in Issue 1.

9.4 Command-Line Interfaces

This section defines the utilities that provide XCDE execution management services.

NAME

dtaction — invoke an XCDE action with specified arguments

SYNOPSIS

```
dtaction [–contextDir context_dir] [–execHost host_name]
[–termOpts terminal_arguments] [–user user_name] action_name
[action_arg] ...
```

DESCRIPTION

The *dtaction* utility allows applications or shell scripts, which are otherwise not connected into the XCDE development environment, to invoke action requests.

The action called *action_name* is invoked with the *action_arg* provided on the command line. A single *action_name* is required; the user may provide any number of *action_arg*s. Interpretation of the *action_name* and *action_arg*s depends on the definition of the action in the action database (see Section 9.5 on page 489). The action may be defined in one of the system action database files, or in one of the user's private action database files.

The *action_arg*s are absolute or relative pathnames of files. The utility passes this list of files on to the specified action.

Error dialogs are posted when the following conditions are detected:

- could not initialise desktop environment
- invalid user or password
- unable to change ID to the desired user
- no action name specified

OPTIONS

The *dtaction* utility does not support the X/Open Utility Syntax Guidelines because it uses the X Window System convention of full-word options. The following options are available:

–**contextDir** *context_dir*
: If the definition of *action_name* does not define a current working directory (see CWD in Section 9.5 on page 489) for command actions, the user can use this option to specify a default directory context.

–**execHost** *host_name*
: The user can use this option to specify an alternative execution host, *host_name*, for a command action. If the action is not a command action, the *dtaction* utility ignores this option. The action is attempted on *host_name* instead of the hosts specified in the action's EXEC_HOST (see Section 9.5 on page 489) specification. An error dialog is posted if it is not possible to invoke the specified action on any eligible host.

–**termOpts** *terminal_arguments*
: This option allows the user to specify arguments intended for the terminal emulator that is provided for command actions that are not of type NO_STDIO. If there are white-space characters in the *terminal_arguments* string, that string must be quoted to protect it from the shell. These arguments are passed unchanged to the terminal emulator. The user must ensure that they are reasonable; in particular, *terminal_arguments* does not allow the argument that specifies the command to be run in a terminal emulator window (that is, –**e** for *dtterm*).

–**user** *user_name*
: The –**user** option allows a user to specify a user name. If *dtaction* is not currently running as that user, a prompt dialog collects the indicated user password, or the

root user password. Once a valid password is entered, the *dtaction* utility changes so that it is running as the requested user and then initiates the requested action.

OPERANDS

The following operands are supported:

action_name
The name of the action to be invoked.

action_arg
The absolute or relative file names of files.

STDIN

Not used.

INPUT FILES

The input files named as *action_arg* arguments are absolute or relative names of files.

The action database files found on *DTDATABASESEARCHPATH* conform to the format specified in Section 9.5 on page 489.

ENVIRONMENT VARIABLES

The following environment variable affects the execution of *dtaction*:

DTDATABASESEARCHPATH
A comma-separated list of directories (with optional host: prefix) that tells the action service where to find the action databases.

RESOURCES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The *dtaction* utility writes diagnostic error messages to standard error, which is redirected to **$HOME/.dt/errorlog**.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.

>0 An invocation error was detected.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

SEE ALSO

Section 9.5 on page 489, *dtterm*, *dtaction*.

CHANGE HISTORY

First released in Issue 1.

9.5 Data Formats

XCDE actions define the behaviour of icons, front panel controls and operations on data objects. Actions are defined in a set of text files with the **.dt** suffix. Each action definition consists of the word ACTION followed by an action name and a list of Field and Value pairs (one per line) on lines by themselves and enclosed in brackets.

9.5.1 Action File Syntax

The general syntax of the actions files is as follows:

```
set DtDbVersion=version_number
set VariableName=variable_value

ACTION  action_name
{
       # Comment
       FieldName field_value
       FieldName field_value
         .
         .
         .
}
```

These text files may also contain data typing information as described in Section 8.4 on page 453. (See Section 8.4.1 on page 453 for the general syntax and location of the actions and data types database.)

9.5.2 Classes of Actions

Actions are of one of the following classes: command actions, ToolTalk message actions or map actions. These action classes are described in the following sections.

9.5.2.1 Command Actions

Command actions are identified by a TYPE COMMAND field. This field defines an execution string to invoke and a set of related information, such as the current working directory for the command and the host where the command should be executed. The following field names are unique to command actions: EXEC_STRING, EXEC_HOST, CWD, WINDOW_TYPE and TERM_OPTS.

9.5.2.2 ToolTalk Message Actions

ToolTalk message actions are identified by a TYPE TT_MSG field. This field defines a ToolTalk message to be sent. The following fields are unique to ToolTalk message actions: TT_CLASS, TT_SCOPE, TT_OPERATION, TT_FILE, TT_ARGn_MODE, TT_ARGn_VTYPE, TT_ARGn_REP_TYPE and TT_ARGn_VALUE.

9.5.2.3 Map Actions

Map actions are identified by a TYPE MAP field. This field does not define any specific behaviour; instead, this field specifies a different action name that should be invoked in place of the original action. Multiple map actions can be chained together, but the chain must ultimately terminate in a non-map action. The following field is unique to map actions: MAP_ACTION.

9.5.2.4 Common Fields

In addition to the unique action fields listed above, all actions support the following fields: LABEL, ICON, DESCRIPTION, ARG_CLASS, ARG_MODE, ARG_TYPE, ARG_COUNT and TYPE.

9.5.2.5 Keywords

The value string for certain action fields may reference special keywords enclosed within percentage character (%) delimiters. These keywords are evaluated when the action is invoked and replaced with the appropriate value. In fields that do not evaluate keywords, the keyword is taken literally. The valid keywords are:

%DatabaseHost%
: The name of the host where the action definition file is located. This hostname is specified by the host portion of the *host:/path* searchpath used to find the action.

%DisplayHost%
: The name of the host where the X server displaying the XCDE session is running.

%LocalHost%
: The name of the host where the application invoking the action is executing.

%SessionHost%
: The name of the host where the controlling login manager (*dtlogin*) runs.

9.5.2.6 Argument References

Arguments passed to actions can be referenced in certain action fields using special argument keywords enclosed within percent character (%) delimiters. These argument keywords are evaluated when the action is invoked and replaced with the appropriate value. In fields that do not evaluate keywords, the keyword is taken literally. The valid argument keywords are:

%Arg_*n*%
: The *n*th (starting with 1) argument of the action. If the action was invoked with fewer than *n* arguments, the value of the keyword is NULL.

%Args%
: All remaining arguments of the action. If any arguments of the action have already been referenced within this field by an %Arg_*n*% keyword, those arguments are not referenced a second time by %Args%.

%"*prompt*"%
: Prompt the user for a value, using *prompt* as the label of a text field.

%Arg_*n*"*prompt*"%
: The *n*th (starting with 1) argument of the action. If the action was invoked with fewer than *n* arguments, prompt the user for a value using *prompt* as the label of a text field.

If a keyword references the name of a file argument, the value of the keyword is expanded to an absolute pathname prior to substitution. In addition, if the file name is to be passed to a remote system, the file name is first mapped appropriately (see *tt_file_netfile*( ) and *tt_netfile_file*( )).

If the keyword references a buffer argument, the buffer data is placed in a temporary file and the name of the temporary file is substituted, as described above. Some action fields provide direct support for data buffers and do not require use of a temporary file. This behaviour is noted in the description of the appropriate fields.

If the keyword references a string obtained from the user, it is treated as a simple string and the value substituted without any transformation.

Argument references can be forced to be treated as file names or simple strings by using the **(File)** or **(String)** qualifier immediately after the opening % of the keyword. For example:

```
%(String)Arg_n%
%(File)"prompt"%
```

If an action is invoked with more than one argument, and the action definition only references one or zero arguments, the action is iteratively invoked with each of the supplied arguments. If the action definition references more than one argument, any extra arguments are ignored.

9.5.2.7 Action Selection

Multiple actions can be defined with the same name. When the action is invoked, the appropriate action definition is chosen based on the number and class of arguments supplied. For example, the **Open** action may invoke *dtpad* if a text file is supplied as an argument, or it may invoke *dticon* if a bitmap file argument is supplied. The ARG_COUNT, ARG_CLASS, ARG_MODE and ARG_TYPE fields specify the number, mode and types of arguments that are accepted by a particular action. Because these fields can have shell pattern-matching values such as *, it is possible that the action database contains multiple actions that have the same name and are all capable of accepting the arguments that are supplied. In this case, the following precedence rules are used to choose a single action definition to invoke:

- Actions with more specific attribute values take precedence over more general attribute values.
- For the ARG_COUNT field, an exact numerical value (*N*) is more specific than a less-than range (<*N*). A less-than range (<*N*) is more specific than a greater-than range (>*N*). And a greater-than range (>*N*) is more specific than a shell pattern-matching character (*).
- For the ARG_CLASS and ARG_TYPE fields, a single item is more specific than a list of items and a list of items is more specific than a shell pattern-matching *.
- For the ARG_MODE field, **w** (writable) and **!w** (not writable) are more specific than a shell pattern-matching *.
- The fields have the following precedence, from high to low: ARG_CLASS, ARG_TYPE, ARG_MODE, ARG_COUNT.
- If two action definitions have equal specificity, the action definition appearing first in the database load order takes precedence. Database directories are loaded in the order specified by the *DTDATABASESEARCHPATH* environment variable, and are loaded in the collation order of their file names.

9.5.2.8 ARG_CLASS Field

The ARG_CLASS field is optional for all types of actions. This field specifies the class of arguments the action accepts. If an action is invoked with more than one argument, the class of only the first argument is checked against the value of the ARG_CLASS field. The valid values for this field are:

BUFFER	The action accepts arguments that are blocks of data held in memory.
FILE	The action accepts arguments that are file names.
*	The action is defined for all classes of arguments.

A comma-separated list of valid values is also allowed and specifies that the action accepts arguments of any of the listed classes.

If an action is defined to accept a buffer argument, yet the implementation of the action requires a file name, the buffer is automatically converted into a temporary file for the action to use. See the description of the **DtTmpDir** resource (Section 9.5.3 on page 497) for information on configuring the location of these temporary files.

Keywords are not evaluated in the ARG_CLASS field. The default value of the ARG_CLASS field is ∗.

9.5.2.9 ARG_COUNT Field

The ARG_COUNT field is optional for all types of actions. The ARG_COUNT field specifies the number of arguments that the action accepts. The valid values for this field (where *N* denotes any non-negative integer) are:

N The action accepts exactly *N* arguments.

<*N* The action accepts any number of arguments less than *N*.

>*N* The action accepts any number of arguments greater than *N*.

∗ The action accepts any number of arguments.

Keywords are not evaluated in the ARG_COUNT field. The default value of the ARG_COUNT field is ∗.

9.5.2.10 ARG_MODE Field

The ARG_MODE field is optional for all types of actions. This field specifies the mode of arguments the action accepts. If an action is invoked with more than one argument, the mode of only the first argument is checked against the value of the ARG_MODE field. The valid values for this field are:

w The action accepts arguments that writable by the user.

!w The action accepts arguments that are not writable by the user.

∗ The action is defined for all classes of arguments.

Keywords are not evaluated in the ARG_MODE field. The default value of the ARG_MODE field is ∗.

9.5.2.11 ARG_TYPE Field

The ARG_TYPE field is optional for all types of actions. This field specifies the types of arguments the action accepts. If the action is invoked with more than one argument, the type of only the first argument is checked against the value of this field. Valid values for this field are ∗ (all data types are accepted), a single data type name or a comma-separated list of data types. The set of valid data types are those defined by DATA_ATTRIBUTE records in the data typing database. (See Section 8.4 on page 453 for more information.)

Keywords are not evaluated in the ARG_TYPE field. The default value of the ARG_TYPE field is ∗.

9.5.2.12 CWD Field

The CWD field is optional for all types of actions. This field specifies the current working directory to be used when the execution string is invoked. Valid values include any absolute pathname. If this field is not specified, the current working directory for the execution string is determined by the following:

- If the application invoking the action specifies a current working directory, that directory is used.
- If arguments are supplied to the action and the first argument is a directory, that directory is used.
- If arguments are supplied to the action and the first argument is a file, the directory where the file is located is used.
- The current working directory of the application invoking the action is used.

Keywords are not evaluated in the CWD field.

9.5.2.13 DESCRIPTION Field

The DESCRIPTION field is optional for COMMAND actions. This field specifies a textual description of the action that is suitable for presentation to a user requesting information about the action. The description should contain no formatting information such as tab or newline characters. The application that presents the information to the user formats the information.

Keywords are not evaluated in the DESCRIPTION field. There is no default value for the DESCRIPTION field.

9.5.2.14 EXEC_HOST Field

The EXEC_HOST field is optional for COMMAND actions. This field specifies the host where the execution string should be invoked. Valid values for this field include actual hostnames, as well as any of the hostname keywords. If a comma-separated list of hostnames is provided, execution is attempted on each of the hosts in the order specified until execution succeeds.

Keywords are evaluated in the EXEC_HOST field. The default value of the EXEC_HOST field is **%DatabaseHost%,%LocalHost%**. (See the description of the **ExecutionHosts** resource in Section 9.5.3 on page 497 for information on how to change this default value.)

9.5.2.15 EXEC_STRING Field

The EXEC_STRING field is required for COMMAND actions. This field specifies an execution string to be invoked. The string is parsed using the same quoting rules as defined by *sh* in the X/Open CAE Specification, **Commands and Utilities**, **Issue 4, Version 2**; however, the execution string is not automatically passed to any shell. Therefore, if the execution string requires shell features such as redirection of standard input, redirection of standard output, or pipes, the appropriate shell must be specified explicitly in the execution string. For example:

```
EXEC_STRING   sh –c 'ls –l | more'
```

Keywords are evaluated in the EXEC_STRING field. There is no default value for the EXEC_STRING field.

9.5.2.16 ICON Field

The ICON field is optional for all types of actions. This field specifies the name of an icon that represents the action.

Icons are found by using the standard XCDE icon search path, so the value can be either an absolute pathname (for example, **/foo/icons/myicon.bm**), a relative pathname (for example, **icons/myicon.bm**) or a partial filename (for example, **myicon**). Partial filenames are preferred because they allow the XCDE icon search path to find the icon with the optimum size and depth for the current environment.

Keywords are not evaluated in the ICON field. The default value of the ICON field is **Dtactn**. (See the description of the **ActionIcon** resource in Section 9.5.3 on page 497 for information on how to change this default value.)

9.5.2.17 LABEL Field

The LABEL field is optional for all types of actions. This field specifies a user-visible label for the action. When actions are presented to the user, the localised LABEL field is used to identify the action instead of the non-localised action name.

Keywords are not evaluated in the LABEL field. The default value of the LABEL field is the name of the action.

9.5.2.18 MAP_ACTION Field

The MAP_ACTION field is required for MAP actions. This field specifies the name of an action that should be invoked in place of the current action. The specified action is invoked with the same set of arguments that were passed to the original action.

Keywords are not evaluated in the MAP_ACTION field. There is no default value for the MAP_ACTION field.

9.5.2.19 TERM_OPTS Field

The TERM_OPTS field is optional for COMMAND actions. This field specifies command-line options that are passed to the terminal emulator for all COMMAND actions that are terminal based. (That is, any COMMAND action other than those that specify WINDOW_TYPE NO_STDIO.) These command-line options are typically used to specify a unique terminal-window geometry, font, colour or title.

The value of the TERM_OPTS field must be an option string in a form the terminal emulator supports and it must only affect the appearance of the terminal window. For example, options such as **–e**, which affect the behaviour of the terminal window, must not be used.

Keywords are evaluated in the TERM_OPTS field. The default value of the TERM_OPTS field is

```
–title action_label
```

where *action_label* is the LABEL field for the action. See *dtterm* for the meaning of **–title**.

9.5.2.20 TT_ARGn_MODE Field

The TT_ARGn_MODE field is optional for TT_MSG actions. This field specifies the value of the ToolTalk mode attribute for the *n*th message argument, where *n* is zero for the first message argument. The valid values for this field are: TT_IN, TT_INOUT and TT_OUT.

(See **<Tt/tt_c.h>** in Section 6.3 on page 318 for a description of these values.)

Keywords are not evaluated in the TT_ARGn_MODE field. There is no default value for the TT_ARGn_MODE field.

9.5.2.21 TT_ARGn_REP_TYPE Field

The TT_ARGn_REP_TYPE field is optional for TT_MSG actions. This field specifies the representation type of the *n*th ToolTalk message argument, where *n* is zero for the first message argument. The valid values for this field are:

TT_REP_UNDEFINED
: If TT_ARGn_VALUE references a buffer argument, the representation type is a buffer; otherwise, it is a string.

TT_REP_INTEGER
: The representation type is an integer.

TT_REP_BUFFER
: The representation type is a buffer.

TT_REP_STRING
: The representation type is string.

Keywords are not evaluated in the TT_ARGn_REP_TYPE field. The default value of the TT_ARGn_REP_TYPE field is TT_REP_UNDEFINED.

9.5.2.22 TT_ARGn_VALUE Field

The TT_ARGn_VALUE field is optional for TT_MSG actions. If there is no corresponding TT_ARGn_MODE field, the TT_ARGn_VALUE field is ignored. If there is a TT_ARGn_MODE field, the TT_ARGn_VALUE field specifies the value of the *n*th ToolTalk message argument, where *n* is zero for the first message argument. If there is a TT_ARGn_MODE field with no corresponding TT_ARGn_VALUE field, the value of the *n*th ToolTalk message argument is set to NULL.

The value of the TT_ARGn_VALUE field must be a single string or action argument. Keywords that reference a single action argument are evaluated in the TT_ARGn_VALUE field, however %Args% is not allowed as it references multiple action arguments. There is no default value for the TT_ARGn_VALUE field.

9.5.2.23 TT_ARGn_VTYPE Field

The TT_ARGn_VTYPE field is required to accompany any TT_ARGn_MODE fields in TT_MSG actions. This field specifies the value of the ToolTalk vtype attribute of the *n*th message argument, where *n* is zero for the first message argument. If this field references an argument keyword, the *MEDIA* attribute of the specified argument is used. If the *MEDIA* attribute is not defined, the DATA_ATTRIBUTE name of the data type is used.

Keywords are evaluated in the TT_ARGn_VTYPE field. There is no default value for the TT_ARGn_VTYPE field.

9.5.2.24 TT_CLASS Field

The TT_CLASS field is required for TT_MSG actions. This field specifies the value of the ToolTalk class message field. The valid values for this field are:

TT_NOTICE
: The action defines a ToolTalk notification message.

TT_REQUEST
The action defines a ToolTalk request message.

Keywords are not evaluated in the TT_CLASS field. There is no default value for the TT_CLASS field.

9.5.2.25 TT_FILE Field

The TT_FILE field is optional for TT_MSG actions. This field specifies the value of the ToolTalk file message field. The value of this field must be a single file name and can either be a specific file name (for example, **/tmp/foo**) or an argument keyword (for example, %Arg_*1*%). %Args% is not allowed because it references multiple action arguments. If an argument keyword is specified and the corresponding argument is not a file (that is, it is a buffer), the action invocation fails.

Keywords are evaluated in the TT_FILE field. There is no default value for the TT_FILE field; if it is not set, the file attribute of the ToolTalk message is set to NULL.

9.5.2.26 TT_OPERATION Field

The TT_OPERATION field is required for TT_MSG actions. This field specifies the value of the ToolTalk operation message field. Typical values are operations such as *Display* or *Edit* that are defined by the Media Exchange Message Set.

Keywords are not evaluated in the TT_OPERATION field. There is no default value for the TT_OPERATION field.

9.5.2.27 TT_SCOPE Field

The TT_SCOPE field is required for TT_MSG actions. This field specifies the value of the ToolTalk scope message field. (See **<Tt/tt_c.h>** for a description of these values.) The valid values for this field are: TT_BOTH, TT_FILE, TT_FILE_IN_SESSION and TT_SESSION.

Keywords are not evaluated in the TT_SCOPE field. There is no default value for the TT_SCOPE field.

9.5.2.28 TYPE Field

The TYPE field is optional for COMMAND actions and required for MAP or TT_MSG actions. This field specifies the type of behaviour defined by the action. Valid values for this field are:

COMMAND
The action invokes a command.

MAP The action specifies a different action name to invoke in place of the current action.

TT_MSG
The action defines a ToolTalk message to be sent.

Keywords are not evaluated in the TYPE field. The default value of the TYPE field is COMMAND.

9.5.2.29 WINDOW_TYPE Field

The WINDOW_TYPE field is optional for COMMAND actions. This field specifies the type of windowing support the execution string requires. Valid values for this field are:

NO_STDIO
: No windowing support is required. This value is appropriate for execution strings that have no output or are X Windows applications.

PERM_TERMINAL
: The execution string requires a terminal window. When the execution string exits, the terminal window is left open until the user explicitly closes it. This value is appropriate for applications that write their output to standard output and then terminate, such as *ls*.

TERMINAL
: The execution string requires a terminal window. When the execution string exits ''quickly'' (see the description of the **waitTime** resource), the terminal window is left open to allow the user to view any error messages that were written to standard output or standard error. This value is appropriate for full-screen terminal applications such as the *vi* editor.

Keywords are not evaluated in the WINDOW_TYPE field. The default value of the WINDOW_TYPE field is PERM_TERMINAL.

9.5.3 Resources

The following resources are available to control the behaviour of actions. These resources must be set for the application that is invoking the action. They can be set for all applications that invoke actions by omitting the application name or class name.

X11 Resources That Modify Action Behaviour			
Name	**Class**	**Type**	**Default**
actionIcon	**ActionIcon**	**string**	''Dtactn''
dtEnvMap-ForRemote	**DtEnvMap-ForRemote**	**string**	''DTAPPSEARCHPATH: DTHELPSEARCHPATH: DTDATABASESEARCHPATH: XMICONSEARCHPATH: XMICONBMSEARCHPATH''
dtexecPath	**DtexecPath**	**string**	''/usr/dt/bin/dtexec''
dtTmpDir	**DtTempDir**	**string**	''$HOME/.dt/tmp''
executionHost-Logging	**ExecutionHost-Logging**	**string**	''False''
executionHosts	**ExecutionHosts**	**string**	''%DatabaseHost%, %LocalHost%''
localTerminal	**LocalTerminal**	**string**	''dtterm''
remoteTerminals	**RemoteTerminals**	**string**	None
waitTime	**WaitTime**	**number**	3

actionIcon
: Specifies the default value of the ICON field for actions that do not define the field. The default value of this resource is **Dtactn**.

dtEnvMapForRemote

Specifies a colon-separated list of environment variables names. Each variable contains a colon-separated list of pathnames to be mapped for remote actions (see *tt_file_netfile*( ) and *tt_netfile_file*( )).

Only environment variables in the user's current environment are mapped.

If a pathname contains substitution characters, only the portion of the path up to the first percent character is mapped, with the remaining portion appended to the resulting mapped portion. For example, if *NLSPATH* is set to **/system/nlslib/%L/%N.cat**, it maps to **/net/host/system/nlslib/%L/%N.cat**.

dtexecPath

Specifies the location of the *dtexec* command that is used for terminal-based actions. The default value is **/usr/dt/bin/dtexec**.

dtTmpDir

Specifies the full pathname of the directory to be used for holding temporary files created during action invocation. The directory must be visible to remote hosts used for action execution.

executionHostLogging

Turns on and off detailed logging to the user's **$HOME/.dt/errorlog** of action invocation events. The default value is False, which disables logging. Logging is enabled if this resource is set to True.

executionHosts

Specifies the default value of the EXEC_HOST field for COMMAND actions that do not define the field. The default value is **%DatabaseHost%,%LocalHost%**.

localTerminal

Specifies an alternative terminal emulator for terminal-based actions that execute locally. Any terminal emulator specified by this resource must support the **–title** and **–e** options as described in *dtterm*. The default value is *dtterm*.

remoteTerminals

Specifies the terminal emulator to use for terminal-based actions that execute on the named system. The value of this resource is a comma-separated list of the form *host*:*terminal-path* where *terminal-path* is the terminal emulator used when invoking terminal-based actions on *host* host. The default terminal emulator used for any host not specified by this resource is the emulator specified by the localTerminal resource.

waitTime

Specifies the time threshold used for COMMAND actions that specify WINDOW_TYPE TERMINAL. If the command exits in less than *waitTime* seconds, the terminal window is left open. The default value is 3.

9.5.4 Examples

The following action is defined to pipe its argument through the *pr* and *lp* commands:

```
ACTION PrintText
{
       ICON          printer
       DESCRIPTION   Paginate and print a text file to the \
                     default printer.

       ARG_TYPE      Text

       TYPE          COMMAND
       EXEC_STRING   sh –c 'pr %Arg_1"File to print:"% | lp'
       WINDOW_TYPE   NO_STDIO
}
```

The following action defines that *Open* on Text files use the EditText action:

```
ACTION Open
{
       ARG_TYPE      Text
       TYPE          MAP
       MAP_ACTION    EditText
}
```

The following action is defined to send the ToolTalk Display request message for non-writable objects:

```
ACTION Display
{
       ARG_CLASS           BUFFER
       ARG_MODE            !w

       TYPE                TT_MSG
       TT_CLASS            TT_REQUEST
       TT_SCOPE            TT_SESSION
       TT_OPERATION        Display
       TT_ARG0_MODE        TT_IN
       TT_ARG0_VTYPE       %Arg_1%
       TT_ARG0_VALUE       %Arg_1%
}
```

9.5.5 Application Usage

Errors encountered when loading database files are written to the user's **$HOME/.dt/errorlog**. Errors encountered in the value of an action field cause the field to be rejected. If the field is a required field, the entire action definition is rejected. Errors encountered when an action is invoked cause an error dialog to be displayed to the user.

Index