Carnegie Mellon University
School of Computer Science / Robotics Institute

Task Control Architecture

Programmer's Guide to Version 8.0

Reid Simmons • Richard Goodwin • Christopher Fedor • Jeff Basista

Manual version: August 1995,
Including release notes through TCA Version 8.3

About this manual…

Abstract

This manual is a programmer's guide to using the Task Control Architecture (TCA), a general-purpose framework for designing and coordinating distributed mobile robot systems. TCA is designed specifically for robots that operate in dynamic and uncertain environments and have multiple, complex tasks to accomplish. It provides facilities to support interprocess communication, hierarchical task decomposition, synchronization and scheduling of subtasks, resource management, execution monitoring and error recovery.

Credits

The Task Control Architecture was designed by Reid Simmons, a Research Scientist in the School of Computer Science at Carnegie Mellon University. See the "Selected Bibliography" (next column) for references to academic papers on the development and use of TCA.

The primary implementors of TCA were Christopher Fedor, Reid Simmons and Richard Goodwin, although contributions were made by other members of Reid Simmons' research group.

Manuals for TCA up to version 6 were written by Reid Simmons and Christopher Fedor. The manual for TCA version 6, which is a substantial revision of earlier versions, was written by Jeff Basista. He also designed the page layouts and text formats, using FrameMaker. Updates for version 7 were done by Reid Simmons.

Contacts

There is a TCA users mailing list, tca-users@cs.cmu.edu. This mailing list provides information about the latest releases, features and bug fixes for TCA. To become a member of the mailing list, send email to tca-request@cs.cmu.edu.

Questions about TCA and suggestions for future releases may be addressed to the tca-users mailing list, or to:

Reid Simmons
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15213-3891

If you send a suggestion for the manual or a correction to it, please be sure to specify the manual version, which is printed on the cover page. If you have a question about TCA, be sure to include the version number, which is printed when the central server is started.

Obtaining TCA Code

TCA is available via anonymous ftp. Login to ftp.cs.cmu.edu as "anonymous" and use your mail address as the password. "cd" to "project/TCA". That directory contains tarred and compressed copies of the latest TCA release including this manual and detailed instructions on installing TCA on your machine.

For example, the file tca-8.0.3.tar.Z contrains release 8.0.3. Install tca by uncompressing the file ("uncompress tca-8.0.3.tar.Z") and untarring it ("tar xf tca-8.0.3.tar") to create a directory named "tca". In the "tca/src/INSTALL" file, you will find the detailed instructions on how to compile and use TCA on your machine.

Selected Bibliography

If you would like to read more about TCA, we recommend the following papers:

Simmons, Reid; Lin, Long Ji; and Fedor, Chris. "Autonomous Task Control for Mobile Robots." In *Proceedings of IEEE Symposium on Intelligent Control*. Philadelphia, PA: September 1990.

Simmons, Reid. "An Architecture for Coordinating Planning, Sensing, and Action." In *Proceedings of DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*. San Diego, CA: November, 1990. 292-7.

Simmons, Reid. "Concurrent Planning and Execution for Autonomous Robots." *IEEE Control Systems* 12.1 (1992): 46-50.

Simmons, Reid. "Monitoring and Error Recovery for Autonomous Walking." In *Proceedings of IEEE International Workshop on Intelligent Robots and Systems*. July, 1992. 1407-12.

Simmons, Reid. "Structured Control for Autonomous Robots." In *IEEE Transactions of Robotics and Automation*. Feb, 1994.

Release Notes

These release notes are for versions of TCA subsequent to 7.3. In case of conflicts between the release notes and the manual, the information contained in the release notes takes precedence.

TCA Version 7.4 Release Notes

This version includes three major additions/changes over previous versions, in addition to numerous minor bug fixes.

1. In previous versions, tcaFreeData() and tcaFreeReply() could not handle situations where the data passed was a null pointer, or if the associated message format was null. This has been changed. The following syntax is now always valid, and is the recommended way to define message handlers:

```
void fooHandler (TCA_REF_PTR ref, void *data)
{ FOO_PTR fooData;
 ...
 fooData = (FOO_PTR)data;
 ...
 tcaSuccess/tcaFailure/tcaReply
 tcaFreeData(tcaReferenceName(ref), data);
}
```

2. In previous versions, message wiretaps that were goals or commands were placed directly under the root of the task tree, regardless of which message they were tapping. This could cause problems: for instance, if the message being tapped was killed, the wiretaps would be unaffected (since they were not in the subtree of the message being killed).

Now, the "-t" command line option causes goal and command message taps to be added as children of the message that was tapped. Thus, when the subtree under a tapped message is killed, so will all the associated wiretap mesages. However, the wiretap is in general not visible when using the TCA utilities to trace a task tree: for example, tcaFindLastChild(), tcaFindNextChild(), etc., will never return a wiretap message. The only exceptions are that tcaFindChildByName() will return a wiretap message of the appropriate name, and given a wiretap message reference, tcaFindParentReference will return the message that was tapped (the same information can be gotten by using tcaFindTappedReference(), which is more general since it works even if the wiretap message is not part of the task tree -- e.g., it is an "inform" message)

To allow time for users to make any modifications to their systems necessitated by this change, the default behavior is still to place tap messages under the root node of the task tree. In subsequent versions, however, the default behavior will change to placing wiretaps under the tapped message, and the "-t" option will be reserved for specifying the current behavior.

3. In previous versions, named formatters were expanded immediately upon registration. This could cause problems if a named formatter in turn used a formatter defined by another module. The solution was either to start modules in a given order, or to register all component formatters of a named formatter in each module. This has been changed in version 7.4. Now, named formatters are expanded only when they are needed (i.e, the first time a message is sent that uses the formatter). As long as systems register all their named formatters before calling tcaWaitUntilReady(), this change will enable developers to register named formatters only once, and the formatters will be correctly propogated to other modules.

TCA Version 7.5 Release Notes

This version fixes a major bug in the way conversion from big to little Endian machines occurred. In the course of fixing this bug, several changes were made to the code having to do with this conversion. As a result, the code was cleaned up, and several extraneous bcopy's were deleted. This should result in marginally faster code.

The data transfer code has been tested somewhat extensively, but there is always the possibility that some cases have gone undetected. If you notice any instances where data does not seem to be transferred correctly to/from Intel or pmax machines, please contact the TCA maintainers.

TCA Version 7.6 Release Notes

This version has three significant additions.

1. Two new functions are added: tcaIgnoreLogging(char *msgName) and tcaResumeLogging(char *msgName). tcaIgnoreLogging() will not log message "msgName" if the logging option in central includes the "i" switch (eg, -lmdsi). This can be useful to keep the log file of manageable size if the message is sent with high frequency. tcaResumeLogging() has the obvious effect of allowing the message to be logged, regardless of the status of the "i" switch (note that if the "i" switch is not in use, all messages will be logged, regardless of their "ignore" status).

2. Conditional compilation switches have been added for a MacIntosh (specifically, MPW-C) version of TCA. This runs using the GUSI socket library, and is currently being tested. The rest of the code needed to run a MacIntosh version of TCA will be released shortly. If you have a pressing need for such a Mac version of TCA, contact the TCA maintainers.

3. There is a very rudimentary command interface on the central server. you can enter command line options while the server is running. One option per line. In addition to the usual options, it takes "quit" and "help". The interface is most useful for turning logging on and off during a tca run.

TCA Version 7.7 Release Notes

Summary:

1) Added Global variables to TCA.
2) Added utilities to limit the number of pending messages
3) Added support for compiling on sgi machines. "makefiles" still needs some cleaning up. See src/INSTALL file for instructions on compiling.

4) Better error checking. TCA now checks for a valid connection to central before trying to send messages.
5) Fixed a bug that allowed central to send messages to modules even before they issued a tcaWaitUntilReady() call.

Details:

1. GLOBAL VARIABLES

TCA now supports global variables that can be set and read atomically. It is intended that this mechanism be used when TCA modules need to share system state, such as the current position of a robot. Procedures are available to set and query the value of a variable. In addition, modules can be notified when the value of a variable changes.

Global variables can be used to de-couple the timing constraints between producers and consumers of information. For example, one module may be keeping track of the position of the robot using dead reckoning. It might be part of the robot's control loop and produce new information frequently. Other modules such as planners, may need this information, either periodically or at specific times. Under previous versions of TCA, other modules would have to query the dead reckoning module to get the information or the dead reckoning module could broadcast a message whenever the value changed.

If queries are used and many modules require position information, then the dead reckoning module will spend a lot of time replying to queries. Broadcast messages can be used to reduce the load on the dead reckoning module. Using broadcast messages, the dead reckoning module needs only send out one message every time the position changes. Modules interested in the position can register handlers for the message and be informed whenever it changes.

The problem is that some modules may only be interested in getting position information at specific times or less frequently that the rate at which it changes. Using broadcast messages, these modules must still handle every broadcast messages that is sent. Using global variables can reduce the overhead. Modules can simply get the value of the variable whenever they need it. The dead reckoning module still only needs to send a single message when the position changes.

If a module is interested in keeping track of the current position as often as it changes, then the module can have TCA "watch" the variable and be notified whenever it is set. If a module is interested in getting the value periodically (say every 2 seconds), then it can set up a polling monitor to query the value every 2 seconds.

GLOBAL VARIABLE FUNCTIONS

void tcaRegisterVar(const char* name, const char* format)

Register a variable with name that has the given data format. The format string uses the same syntax as that for messages.

void tcaSetVar(const char* name, const void* value)

Set the value of a global variable. This action happens asynchronously and is not constrained by commands and goals in the task tree. The value is stored in the central server, so this action only takes a single

void tcaGetVar(const char* name, void* value)

Get the current value of the variable. Blocks until the value is returned.

void tcaGetSetVar(const char* name, void* setValue, void* getValue)

Get the current value of the variable and then set it to the given value. This is the "test and set" function.

void tcaWatchVar(const char *name, const char *format, TCA_HND_FN watchFn)

Watch the variable given by name and call the watchFn with the new value whenever it changes.

2. LIMITING PENDING MESSAGES

TCA limits the number of messages being handled by a module (actually, a resource) at any one time. This is called the resource's capacity, and is usually set to 1 (only one message active at a time). Additional messages destined for that module/resource are queued, pending resource availability. Messages are queued in a first-in first-out manner.

If messages are being sent faster than a module can handle them, the queue of pending messages can grow quite large. Occasionally, one is interested in only the most recent messages, and it is safe to ignore older pending messages. TCA now includes mechanisms for producing the behavior of saving only the most recent messages destined for a module/resource. One can request TCA to maintain only the N most recent messages sent to a given resource, or the N most recent messages with a given name. One can limit any class of message, but it is not recommended for use with query-class messages.

LIMIT PENDING FUNCTIONS

void tcaLimitPendingResource (const char *resName, int limit)

Allow only "limit" pending messages for the resource "resName". Deletes messages in a FIFO manner to maintain this constraint. This constraint applies to *all* messages sent to the given resource. The module name (the name used in tcaConnectModule()) is used to refer to the module's default resource.

void tcaLimitPendingMessages (const char *msgName, const char *resName, int limit)

Allow only "limit" pending messages named "msgName" on the resource "resName". Deletes messages in a FIFO manner to maintain this constraint. The module name (the name used in tcaConnectModule()) is used to refer to the module's default resource.

5. DELAY SENDING MESSAGES UNTIL READY

Contrary to popularly held opinion (and statements in the manual), TCA would forward messages to a module even before it issued a tcaWaitUntilReady() call. This has now been fixed, and no messages, other than registration and internal initialization messages, are sent to modules prior to receiving an indication that the module is ready to begin processing messages. A consequent of this is that modules that want to receive messages *must* issue a tcaWaitUntilReady() call before invoking tcaModuleListen() or tcaHandleMessage() (TCA issues a warning if this is not followed). Modules that only *send* messages are not required to call tcaWaitUntilReady(), but it is strongly recommended that all modules include this call following all message and handler registration calls.

TCA Version 7.8 Release Notes

Release 7.8 is mostly a porting and bug fix update . No functionality has changed.

1) Fixed the way global variables are allocated and initialized for vx_works. Modules started from the same shell would share a global variables, causing problems.

2) Changed the logging functions to accept a variable number of arguments.

3) Added support for linux on a 486 machine.

4) Got afs to work for the Dec alpha, and hopefully other vendor OS's

5) Added generic Makefile so it should be easier to compile on machines without gmake.

6) Some small changes to fix bugs with Mach version, espcially sun4_mach.

TCA Version 7.9 Release Notes

Summary:

Release 7.9 is an optimization release. No added functionality. The system just runs a lot faster, in some cases, faster than TCX.

For comparison, I have run a number of tests comparing TCA 7.8, TCA 7.9 and TCX 12.3 (The most recent TCX version I can find).

Test 1:

One process makes 200 queries of another process. The data size of each request message varies from zero bytes to 1 Megabyte. The reply is alway one integer (4 bytes).

All tests were conducted on a sparc 10 running Sunos. All processes where run on the same machine. Using different machines runs up against the limited bandwidth of the network (10 Mbits/s).

Table 1: Test 1

	TCA 7.8		TCX 12.3		TCA 7.9	
Bytes/ msg	msg/s	bytes/s	msg/s	bytes/s	msg/s	bytes/s
0/4*	242	0	562	2249	400	0
16	233	3735	545	8722	370	5921
256	221	56501	508	130134	352	90226
1K	209	213619	472	483341	333	341455
4K	119	485990	256	1046894	264	1081642
16K	53	871264	105	1722838	134	2188539
64K	17	1118213	31	2000986	42	2735001
256K	5	1194023	9	2237433	12	3059980
1M	1	1220855	2	2257506	3	3171188

* TCX version crashed when a null format was used, so I used a single integer.

Comments: For TCA, the maximum message rate has almost doubled and the maximum through put has almost tripled. Comparing TCA and TCX, we see that for very small messages, TCX is faster by 40%. For larger messages, TCA has a higher throughput. For 1Megabyte messages, TCA is 50% faster than TCX. Somewhere around 2K, is the cross over point where the systems have the same message rate and throughput.

Test 2:

Send 200 messages from one process to another, optimizing the program for raw speed using each messaging system.

	TCA 7.8		TCX 12.3		TCA 7.9			TCA 7.8		TCX 12.3		TCA 7.9	
Bytes/ msg	msg/s	bytes/s	msg/s	bytes/s	msg/s	bytes/s	Bytes/ msg	msg/s	bytes/s	msg/s	bytes/s	msg/s	bytes/s
0/4*	412	0	2578	10311	657	0	0/4*	412	0	657	2627	657	0
16	363	5801	2170	34715	582	9317	16	363	5801	507	8111	582	9317
256	327	83705	1414	361880	426	108941	256	327	83705	509	130202	426	108941
1K	280	286760	853	873005	386	394999	1K	280	286760	336	344414	386	394999
4K	143	585518	258	1057488	302	1236209	4K	143	585518	167	685426	302	1236209
16K	58	955469	104	1699761	135	2210402	16K	58	955469	58	950648	135	2210402
64K	18	1156581	29	1913896	42	2775157	64K	18	1156581	17	1083263	42	2775157
256K	5	1218955	8	2013264	12	3068828	256K	5	1218955	4	1160282	12	3068828
1M	1	1229735	2	2140054	3	3199656	1M	1	1229735	1	1202580	3	3199656

* TCX version crashed when a null format was used, so I used a single integer.

Comments: The TCA maximum message rate has increased by 60% and the through put has almost tripled. Comparing TCA and TCX, we again see the same pattern where TCX is faster for messages will little data and TCA is faster for larger messages. This is mostly due to the topology. TCX messages need only make a single hop, while TCA messages go through the central server. The TCX programs were also optimized to block on the read port and not do a select to see which socket/file had available input. Doing a select would slow the message rate down and is needed if the module receives input from more than one other module.

Test 3:

Same as test 2, except using the central routing feature of TCX. This was a recent addition that allows messages to be routed through the central server.

Conclusions:

If you need a very high message rate with little data to be transfered, then TCX has some speed advantages. For significant data, or if synchronization is needed, the TCX speed advantages disappear, and for large messages, TCA 7.9 is the clear winner.

TCA Version 8.0 Release Notes

Summary:

1) Receiver Translates: Messages are sent using the native byte ordering of the sender and the receiver performs automatics conversion, if needed. When messages are passed between machines of the same type, no translation is done, even if the machines don't use the network byte order (For example, i486 machines).

2) Direct Messages: Messages that don't involve the task tree can now be sent directly from one module to another. You can control this on a per resource basis using the tcaDirectResource call or globally using the "-c" option on the central server. Inform and query messages are passed directly whenever they are not being logged or tapped.

3) New central server interface commands. kill: Kill the task tree. Removes all pending and complete messages. close <module>: Close a connection to a module. display : Show the task tree and all other current messages. dump : Print the malloc chain. Useful for debugging central. You have to compile with dbmalloc for this to be available.

4) Emergency memory free routines try to purge some of the pending queues to get enough memory to continue. This is always a last ditch effort, so it may still fail.

5) "Global" removed from some calls. For example, tcaRootNodeGlobal is now tcaRootNode. The old names are retained as defines for compatibility.

6) Reconfigured the TCA tools to separate the log parser from the tview front end. This will allow the parser to be used for other tools.

7) Added the utilities code to create the devUtils and ezx libraries. The device utilities library provides methods for creating and managing interfaces to hardware devices and other "IO" devices like X11, TCA and stdin. Provides more robust error and interrupt handling to improve your TCA programs. EZX is a wrapper to help use X11 to create interfaces. More documentation is still needed for these libraries.

8) Added tcaModuleProvides and tcaModuleRequires. Both take a variable number of pointers to strings that represent the names of capabilities the module provides/requires. By default, a module provides the capabilities with the name of the module and any other resources it creates. A module may also provide other resources. For instance, a simulator module may provide a number of capabilities that replace the capabilities of the real device controllers. There are also versions that take a NULL terminated array of pointers to strings, rather than use a variable number of arguments.

9) The number of modules defaults to 1. Using the tcaProvides and tcaRequires functions allows the central server to only start modules when the modules it depends on are also ready. Specifying a minimum number of modules is less useful now.

10) tcaMessageHandlerRegistered, tcaMessageRegistered: Determine whether a handler, or message, is registered. It can be used to determine whether optional messages should be sent. The result is cached, if a handler/message is registered. Otherwise, central is polled on every call.

11) tcaMaybeExecute, tcaMaybeExecuteWithConstraints, tcaMaybeExecuteAsync: Macros to test to see if a message has a handler registered and execute the message only if the handler is registered.

12) Support for Solaris and Linux operating systems.

13) tcaGetVersionMajor and tcaGetVersionMinor return the version number for tca, so you can tell which version the code was compiled with.

14) TCA_connect, the routine in the device utilities library now also takes provides and requires lists.

15) tcaGetVersionMajor and tcaGetVersionMinor return the version number for tca, so you can tell which version the code was compiled with.

Details: =======

1) There are several advantages to having the receiver translate the byte order for non-network byte order machines. The most obvious one is that it reduces double translation (native->net->native) when data is sent to processes on the same machine or same type of machine. It also allows the sender to do less data copying. When data must be translated from one byte order to another, to be sent, it must be copied. Otherwise, the translation would overwrite the original data. When receiving data, it is OK to overwrite the data buffer, since it is just temporarily needed to hold the incoming bytes.

2) Direct messages.

TCA will take care of establishing and reusing direct connections from one module to another.

To use direct messages as the default, use the -c option on the central server.

For example: "central -c 1"

All inform and query messages will, by default, go through direct connections. If logging or tracing are turned on, then the messages that are logged or traced pass through the central server so that they can be logged or traced.

You can selectively allow direct connections by using the tcaDirectResource call. This specifies that the messages handled by that resource should be sent directly, when possible.

For example, the module resource can be declared direct: " tcaDirectResource("Module B");"

3) Main event loops and Direct messages.

In previous releases of TCA, there was a single socket that connected a module to the central server. With direct connections, a module may have multiple open sockets. If you use the TCA main event loop "tcaModuleListen", then TCA will manage the multiple connections. If you use TCA with X11 or have created your own event loop, you have to listen for activity on multiple sockets.

If you are using X11, then you can use the routine "connectTCA_X11" in the utils/x11Utils.h file. This routine will add inputs and handlers to the X11 application context to allow the X11 main event loop to handle input on the multiple TCA sockets. The single call should be all that is needed.

If you are interfacing to other devices or want more robust operation, I would suggest you use the deviceUtils library of routines. This library has routine for implementing device controllers that connect to the computer via RS232 lines or memory mapping and supports operations like polling and timeouts. It includes routines for handling X11, stdin and TCA connections. It also handles interrupts and broken pipe errors and allows you .

If you "roll your own" main event loops, the call tcaGetConnections returns a pointer to an fd_set that will contain an up to date set of the open TCA connections. In your event loop, you should add these fds to the read mask in your select call. The call to tcaHandleMessage will accept messages on any of the connections and handle requests for new connections.

4) The default message deliver model for TCA is a guaranteed delivery. If a module falls behind, messages will queue up. You can limit the queue by limiting the pending messages. The default limit is infinite.

There is also a TCA routine to register a handler to call when the central server runs out of memory. The default handler use to do nothing. The new default handler tries to purge the message pending queues to free memory in an emergency.

5) Speed Test :

Send 200 messages from one process to another, optimizing the program for raw speed.

All tests were conducted on a sparc 10 running Sunos. All processes where run on the same machine. Using different machines runs up against the limited bandwidth of the network (10 Mbits/s).

TCA

	TCA 7.8		TCA 7.9		TCA 8.0	
Bytes/ msg	msg/s	bytes/s	msg/s	bytes/s	msg/s	bytes/s
0/4*	412	0	657	0	1769	0
16	363	5801	582	9317	1480	23679
256	327	83705	426	108941	1334	341549
1K	280	286760	386	394999	1027	1051431
4K	143	585518	302	1236209	782	3203804
16K	58	955469	135	2210402	402	6581836
64K	18	1156581	42	2775157	130	8528675
256K	5	1218955	12	3068828	40	10377018
1M	1	1229735	3	3199656	10	10831688

TCA Version 8.1 Release Notes

Summary:

This minor release fixes several important bugs.

1) Several important bug fixes:

a) A memory leak was fixed that could cause TCA central to grow without bound.

b) Fixed a problem that occurred when centrally generated messages were queued.

c) Fixed problem that occurred when a module needed to try several times to send a buffer (because the buffer was full). Previously, it would send some of the same parts of the buffer over again.

d) Fixed a problem with sending a structure of doubles from one machine to another machine of the same type.

e) Fixed a problem with direct connections that would cause TCA to crash when a command or goal message was sent from an inform or a query. As part of that fix, all command and goal messages that are sent from informs or queries are now added to the root node of the task tree (since informs and queries are not part of the task tree, previously the command and goal nodes were left unattached to any part of the task tree).

2) Brought the LISP version of TCA back into sync with the C version. Almost all relevant TCA functions (except mainly for the memory management functions) are now available in LISP.

TCA Version 8.2 Release Notes

Summary:

TCA now keeps track of which broadcast messages have handlers registered and only broadcasts a message if there is a handler. This is all done in a manor transparent to the user. You don’t have to change any code to take advantage of this feature.

There is a new function call that a module can use to indicate whether it will listen for tca messages, tcaWillListen(int listen). Calling this function allows module to run more efficiently in some cases.

All tca internal messages have names that start with “tca_”. This will make it easier to distinguish system messages. For compatibility, the old message names are still supported.

The “context” switching has been improved and allows a process to connect to multiple central servers and forward query and command messages as well as broadcast messages. This work was done in preparation for a more generic messaging bridge.

The tcaDevice in the devUtils package has been improved to handle multiple central servers as well. The interface mostly remains the same, except that the internal data structure, TCA_DEV_TYPE, is no longer exported. The only inconsistency is that TCA_disconnect now takes a TCA_DEV_PTR rather than a DEV_PTR.

Replace: TCA_disconnect(TCA_device->dev); With: TCA_disconnect(TCA_device);

The TCA_connect function now returns a TCA_DEV_PTR that can be used on subsequent calls.

The following functions are new or have changed:

void TCA_setCentralHost(const char *machineName); - Set the host name for the central server. Call this before calling TCA_connect to connect to a central, other than the default central.

DEV_PTR TCA_getDev(TCA_DEV_PTR tcaDev); - Function to return the device pointer for a tca device.

BOOLEAN TCA_isConnected(TCA_DEV_PTR tcaDev); - Now takes a TCA_DEV_PTR so it can be used with multiple central servers.

Details:

The central server has a list of all messages and handlers. When a module starts up, it requests a list of broadcast messages that have handlers registered. It then checks this list to see if a message has a handler before broadcasting the message. If the list of broadcast messages with handlers changes (new ones added, old ones removed), then central informs the modules of the new list. This all works transparently to the module programs.

There is one catch. The module programs have to call tcaModuleListen or devMainLoop or call tcaHandleMessage periodically. This is to ensure that the message informing the module of new broadcast handlers is processed. If a module registers a message handler, TCA assumes that it will have to check for this message and will also get the broadcast handler list message. If a module does not register any message handlers, then the module reverts to the original behaviour of always broadcasting messages since it will not have an up to date list of handlers.

The new function call, tcaWillListen(int listen), can be used to explicitly tell tca whether the module will listen for messages. In addition to helping improve the efficiency of broadcast messages, it can also help with logging and direct connections. When the state of logging changes, the central server informs the modules so they can redirect direct messages through the central server so they get logged. Otherwise, the modules do not get updated logging information.

TCA Version 8.3 Release Notes

<u>Nanny</u>

TCA provides many of the facilities necessary for fault tolerant robot systems. However, TCA has no concept of the process control or file system organization required to handle failures at the system level. Using a TCA system usually requires running a number of processes on a number of machines. The user than monitors the processes and restarts them when they crash or a machine reboots.

The organization typically used in robot projects is to start up multiple xterms, connected to appropriate machines. For a typical runtime situation, a half dozen such windows may be necessary. Most people don't know nor wish to know what each of these processes do. Instead, they only care about these processes when they need to control them, and they want to be notified that something has gone wrong. Monitoring the extraneous windows proves to be distracting and confusing in practice.

To address this problems, We've created two programs; nanny runConsole

The idea is to have a single process on each machine (a nanny) that starts up and monitors (baby sits) the processes needed to run the robot. The nanny takes a parameter file which has all the information needed to give the programs the correct arguments and environment variables and knows which programs depend on other programs being run first. Or course, since the nanny starts these programs and manages all the IO, you can't directly see what is going on. That is where runConsole comes in. The runConsole program connects to the nanny server and allows you to interact with any of the running programs (see the output, type new input), on any of the computers. You can start up multiple runConsole programs and each one can connect to processes running on any of the machines.

With complex multi person robot projects, the standard run time set up has typically relied upon people being coached in handling the setup and management of subsystem processes. NANNY is a system for extracting much of this knowledge, codifying it, and then managing the inevitable pitfalls of multiple processes running in the real world (code failure etc).

See the extensive README file in the tools/nanny directory and the example parameter files tca.rc and simulator.rc.

<u>DevUtils</u>

A number of internal improvements have been made to the devUtils library, including some bug fixes. The following functions have been added to the devUtils interface.

devFreeDev: Disconnects the device and frees the memory associated with a device. It can be called from the disconnectHnd. The pointer is invalid after the call, but the memory will not be freed until it is safe to do so.

void devFreeDev(DEV_PTR dev);

devFreeLineBuffer: Free memory for a line buffer.

void devFreeLineBuffer(DEV_LINE_BUFFER_PTR lineBuf);

devSetLineBufferData: routine set the clientData for a line buffer.

void devSetLineBufferData(DEV_LINE_BUFFER_PTR lineBuffer, void *clientData);

devSetLineBufferData: routine set the clientData for a line buffer.

void *devGetLineBufferData(DEV_LINE_BUFFER_PTR lineBuffer);

devConnectDevReceiveOnly: Like devConnectDev, but the device is setup only to listen to the fd and not to send anything.

void devConnectDevReceiveOnly(DEV_PTR dev, int fd);

/* * devConnectDevSendOnly: Like devConnectDev, but the device is setup * only to talk to the fd and not to listen. */

void devConnectDevSendOnly(DEV_PTR dev, int fd);

devSetCloseOnZero: Indicates if a fd is to be closed when the read bit in the select read mask is set, but no characters are available on the port. Typically, you do want to close pipes, but not sockets or real devices like ttys. The default is not to close, but if you have a pipe, you should set this flag so that devUtils does not go into a busy loop.

devSetCloseOnZero(DEV_PTR device, BOOLEAN closeOnZero);

Makefile

The Makefiles have been removed. They were old and confusing. It is much better to use the gnu version of make that will use the GNUmakefiles. If you are really stuck, then try using the Makefile.generic. It is a very stripped down version of the makefile that should run on most machines.

Bridge

We have included a simple bridge that cross registers messages from one central server to another. This allows modules to connected to the second central server to send messages to modules connected to the first central server. A second bridge could be used to send messages in the other direction. Currently, the bridge tries to cross register all the available messages. This is probably not what you would want for any large system. The program is included as an example of how to use devUtils to connect to multiple central servers.

Windows NT

Iain Shigeoka at the University of Kansas has given us an initial port of TCA to windows NT. Please contact rich@cs.cmu.edu if you are interested in being a beta tester for the port or if you want to try the port under windows95 or any other windows version.

Lisp

We fixed a bug where tca was saving pointers to strings rather than copying the strings. This can cause problems when lisp garbage collects. The same changes also makes the C version safer.

Table of Contents

Programmer's Guide to Version 7.3 1

About this manual… 4

Abstract 4

Credits 4

Contacts 4

Obtaining TCA Code 4

Selected Bibliography 4

Release Notes 5

Table of Contents i

Introduction 1

Message passing 3

Sample layout of a TCA-based distributed system 4

The central server 5

Connecting modules 6

Registering messages 7

Registering message data formats 8

Registering your own data format names 9

Number of bytes: an alternative format specifier (C users only) 9

Handling messages 10

Sending messages 11

Query messages 12

Replying to query messages 13

Deferring responses to a message 14

Command messages 14

Inform messages 15

Broadcast Messages 16

Multi-Query Messages 17

Reclaiming data space (C users only) 17

Resource management 19

Building a Resource 20

Locking a resource 20

Reserving a Resource 21

Task management 23

Controlling task execution order 23

Planning vs. execution 24

Goal messages and the creation of task trees 24

Creating placeholder nodes 26

Temporal intervals of planning and execution 27

Specifying Constraints 29

Operations on Task Trees 30

Traversing Task Trees 30

Displaying Task Trees 32

Pruning Task Trees 33

Monitors....................................35

Point Monitors 35

Interval Monitors 36

Polling Monitors 36

Demon Monitors 36

Exception handlers 39

Sending an exception message 40

Registering exception messages and handlers 40

Attaching exception handlers to nodes or messages 40

Handling exceptions 41

An example—docking 42

Wiretaps.................................. 43

Wiretap Conditions 44

TCA and X Windows................ 47

Example programs.................. 49

Index 63

Introduction

The Task Control Architecture (TCA) simplifies building task-level control systems for mobile robots. By "task-level," we mean the integration and coordination of perception, planning and real-time control to achieve a given set of goals (tasks). TCA provides a general control framework, and it is intended to be used to control a wide variety of robots. Although TCA has no built-in control functions for *particular* robots (such as path planning algorithms), it provides control functions, such as task decomposition, monitoring, and resource management, that are common to many mobile robot applications. To date, we know of about a dozen robot systems that have employed TCA, including both indoor and outdoor, autonomous and teleoperated robots.

TCA allows you to construct a distributed system without having to build your own remote procedure call mechanism. At its core, TCA provides a flexible mechanism for passing coarse-grained messages between processes (which we call *modules*). TCA provides orderly access to robot resources so that you don't have to build your own scheduling mechanism. These features are discussed in the first two chapters: "Message passing" and "Resource management."

Using this base of control, you can construct *deliberative* behaviors for your robot. The tools in the "Task Management" chapter allow you to build *task trees*, which specify hierarchical plans of action. You can control the order of planning and execution by applying temporal constraints to task tree nodes. You can also incrementally relax constraints to take advantage of concurrency when needed.

The following chapters, "Monitors," "Exception Handlers," and "Wiretaps," present tools for incrementally adding *reactive* behaviors to your control system without disturbing the basic, deliberative algorithms. Monitors test for some condition—often periodically over some time interval—and specify appropriate action when the condition holds. Exception handlers are invoked when something has interfered with the robot's plans; error recovery strategies are implemented by manipulating the task tree to "re-plan" robot behavior. Wiretaps allow you to "listen" for particular messages in your system; you can specify that some action be taken each time that a certain message is sent.

By providing separate mechanisms for deliberative and reactive behaviors, TCA facilitates incremental development. Building deliberative behaviors, and then adding reactive behaviors as needed, is particularly useful for robots that operate in dynamic or unpredictable environments, because it is impossible to anticipate all possible problems that the robot might encounter. This is one of TCA's key features: separating the "normal" and "abnormal" algorithms of a robot control system and enabling system reliability to be increased in an evolutionary fashion.

Capabilities

The following are key capabilities of autonomous robots; the Task Control Architecture provides support for all of them.

Achieving Goals

The most basic need of a robot is to construct and execute plans to achieve given goals, such as navigating to a particular location or collecting a desired sample. A robot must construct plans based on the current (or projected) environment, available resources, and its other goals and beliefs. It should be able to execute parts of its plans before specifying them completely—for instance, orienting itself to pick up a sample before determining exact grasp points.

Reacting to Environmental Changes

If the robot's battery charge is low, or if its movement uncovers interesting rock formations, the robot should notice the changes in a timely manner and attend to them. To handle such changes, the robot may have to add new sub-goals (e.g., examine and perhaps sample the new formation), suspend current tasks (e.g., stop acquiring the sample and recharge), or re-plan current goals.

Recovering from errors

One type of reactive behavior is noticing when a plan of action is failing (is not meeting expectations). The robot should have general mechanisms for recovering from both execution and plan time errors. For example, if an obstacle appears in the robot's path, it might want to detour around the object, or plan a new path to its goal. Similarly, if the robot planner is unable to find a clear path, it may try re-planning with less restrictive constraints (e.g., allowing more tolerance), or may attend to a different goal altogether.

Coordinating Multiple Tasks

For autonomous robots with many goals but limited resources, it is crucial to prioritize and schedule its various tasks. The robots should make decisions about which goals to attend to by comparing their relative costs, benefits, likelihoods of success, etc. For example, if a robot's batteries are getting low, it might decide to stop what it's doing and recharge. However, if it estimates that it can finish its current sampling task in the next few minutes, it might decide to continue and then recharge.

Communicating with external information sources

No robot will be fully autonomous; interaction with humans is a necessity. The robot should be able to explain its decisions and actions. It also should allow people to add goals and to alter plans and decisions made by the system, and to request assistance when needed. This is especially important for robots, such as planetary rovers, that experience a significant communications delay. To this end, the architecture allows humans to input information on any level of the planning and control hierarchy, including viewing and manipulating autonomously created plans.

Limitations

TCA may not be an appropriate framework for real-time control systems. It is not intended for robots requiring fast (millisecond) reflexes.

Language support

TCA is available in both C and Allegro Common Lisp implementations. It currently runs on the following architectures and operating systems: Sun4 (both SunOS and Mach), SGI, Intel 386 and 486 processors running Mach, PMAX running Mach, NeXT machines, and 68020, 68030 and 68040 processors running VxWorks.

Message passing

A particularly useful model in software engineering is that of the "black box" server program. Such a program provides a small number of related services. If you send a request for a service, along with applicable data, the program will perform the service, report the status (success or failure), and—if applicable—deliver data back to you. You need to know nothing about the internal workings of the server—only what services it provides, what information it requires, and what information it may return, along with the required formats of the information.

An individual server may act as a *module* of a larger system. A system based on such modules has several advantages over monolithic programs.

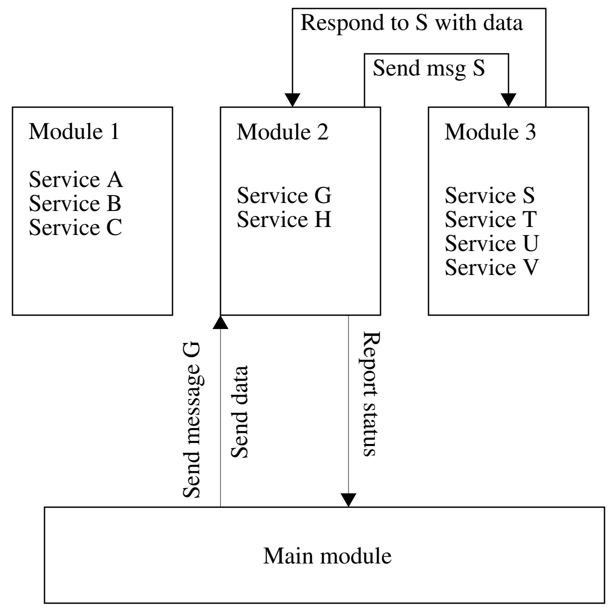
- When functionality and data structures are isolated, you can develop one module in the system without affecting the behavior of others; incremental development and testing is thus simplified.
- Requests handled by different servers can be processed simultaneously.
- If a particular server proves to be a bottleneck, it may be possible to divide up the server's functionality, or to run several identical servers, to increase the available resources.

To build software systems based on this model, you must define the services (functionality) you require and implement appropriate servers, isolating related services in individual programs. For example, suppose you wanted to build a software system that required several groups of services:

- Group 1 (e.g., services that control a robot's legs)
  - service A (e.g., lift leg)
  - service B (e.g., move leg forward)
  - service C (e.g., return current leg position)
- Group 2 (e.g., services that compute paths for the robot)
  - service G
  - service H
- Group 3 (e.g., services that control a robot's scanners)
  - service S
  - service T
  - service U
  - service V

You could construct servers that implement these services. When a "requester" procedure in your system requires a service, it sends a request, and related data if necessary, to the server. We will refer to these requests as *messages*.

The figure below illustrates a few transactions in a very simple distributed system.



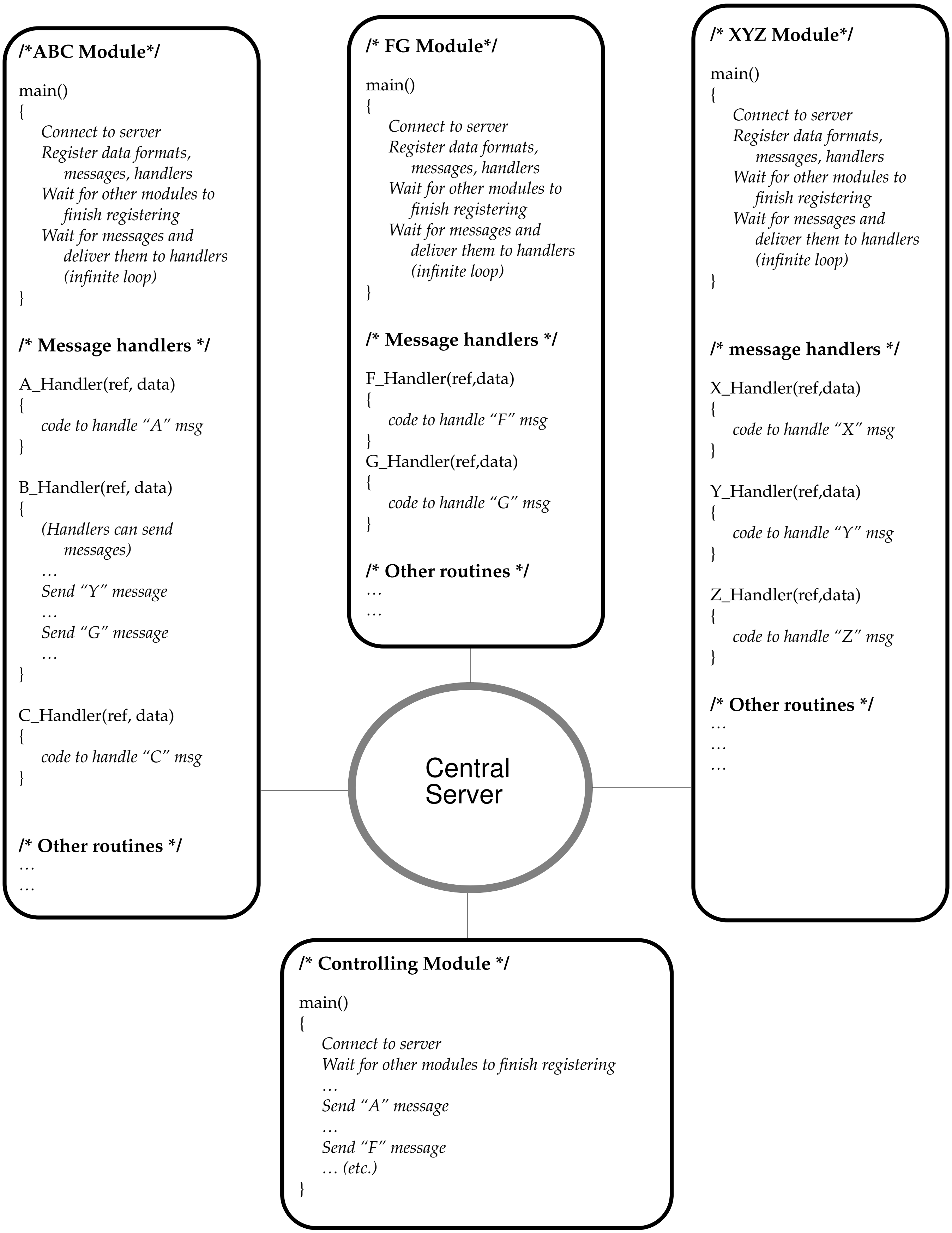
In many systems, a server can also act as a requester; it may require services that it cannot provide itself. In the example above, Module 2 requests service S in order to process the request for service G. In the most general case, any component of the system may need to send messages to any other component.

Unfortunately, building a distributed system normally requires considerable expertise in the interprocess communication (IPC) mechanisms of a particular operating system. You must provide for connections between components that need to communicate. You must ensure that data can be passed transparently between two independent processes.

Matters get more complicated if messages must be handled in a particular order, or if different services make conflicting demands on system resources. You must provide for scheduling and resource management, and neither of these tasks is trivial. You must build something similar to an operating system.

In essence, TCA provides an "operating system" designed specially for task-level robot control. With TCA's message passing routines, you can develop a distributed robot control system without any knowledge of low-level communication software. Basically, you can pass C or LISP data structures between processes (even on different machines) as if

Sample layout of a TCA-based distributed system



you were making function calls. TCA handles all communication using UNIX sockets and TCP/IP.

In order to build a distributed system using TCA's message passing routines, you must:

- Associate each service (physical action, plan generation, information retrieval, etc.) with a *message* that will request the service; for example, you might define messages such as "Move", "PlanPath", and "Current Pose."
- Specify the format of the data required by the handler, and the format of the data that the handler will return, if any. Defining the data structures that get passed between components must be done to define the interface between components.
- Write procedures that will perform the desired services. Each procedure, called a message *handler*, should take only two parameters: the message reference (an identifier that TCA will provide for each message instance), and the data that will be used by the handler.
- Construct small server programs, called *modules*, that contain closely-related handlers, along with their message definitions, internal data structures, and auxiliary routines (if any). These modules must register their messages, handlers, and data formats with the TCA central server. Each module will be a free-standing program running in its own process.

See the opposite page for a sketch of a very simple TCA-based system.

The central server

TCA uses a robot-independent central server module to route and schedule messages. All message traffic passes through the central server.

Before starting any of your modules, you must first start the central sever. The procedure for starting the server is described in the box on the right (see the "Example Programs" chapter for information on where the code resides).

The most basic service that the central server provides is message passing. You can send a message from any module connected to the server, and the server will forward it to the module containing the handler for the message.

You may run more than one server on the same machine, using separate communication ports for each server. Having multiple servers is especially useful for software development —if independent developers must run their TCA servers on the same machine, there is a way to distinguish them.

After messages have been registered, modules can send messages to the central server, and the server will be able to forward each message and its associated data to the module that contains the message handler—the procedure that provides the service.

Starting the central server

% **central** *exp-mods* [**-l** *options*] [**-L** *options*] [**-f** *filename*]

exp-mods: The *expected number of modules* is the number of modules in your system that contain tcaWaitUntilReady() calls. All pending tcaWaitUntilReady() calls return when a total of *exp-mods* calls have been counted by the central server. This is just a minimum number: additional modules may connect to the system at any time.

-l Display logging information on the terminal, with these options:

- **m** Summarize message traffic;
- **s** Report any TCA status messages;
- **t** Specify the time that messages are handled;
- **d** Summarize data associated with messages;
- **i** Ignore registration messages;
- **h** Summarize the handling time for a message;
- **r** Print the message's reference id number
- **p** Print the message's parent id (who sent the message)
- **x** Do not log to the terminal;

Specifying **-l** with no options is shorthand for **-lmstdhrp**.

Options **r** and **r** are most useful in conjunction with the tcaTool (used to view task trees).

If you do not specify **-l**, you will get the default options **-lmsi**. To disable logging to the terminal, you must use **-lx**.

-L Log messages in a file. If you do not specify a filename (with **-f**), one will be generated automatically. You'll be asked to type a description of the session; end it with a blank line. Options are the same as for **-l**, and the default is **-Lx** (no logging to a file.)

-f Use the file *filename* for the **-L** options.

-v Display the server's version information.

-p Specify the server port (default is 1381).

-r try re-sending messages for a module if it crashes;

-h Display a help message that summarizes the command options.

If you use the procedure tcaServerMachine() in a module, you'll need to set the environment variable CENTRALHOST before you start the module. Specify the machine on which the central server is running:

```
setenv CENTRALHOST audrey.learning.cs.cmu.edu
```

The default TCP socket port for the server is 1381. If the desired central server uses a different socket port (i.e., you used the -p option to **central** to start the server), you must provide the port number:

```
setenv CENTRALHOST audrey.learning.cs.cmu.edu:1621
```

If you start a module after making this definition, it would attempt to connect to TCP port 1621 of the host "audrey…"

Connecting modules

To make a module part of the robot system, use tcaConnectModule() (described below). A module can connect at any time the central server is running. Once the connection is established, a module should register its message handlers, along with the messages themselves and the corresponding data formats. (There are also other items that can be registered, and these will be discussed in subsequent chapters.)

When a module has sent all of its registration data, it should call tcaWaitUntilReady(), which will pause the module until the central server has determined that all modules have finished registration and messages can be forwarded correctly. To make this determination, the server counts tcaWaitUntilReady() calls until the count reaches the *expected number of modules*, a number that you provided as a required command line argument when you started the central server (see box on previous page).

If a module only processes requests from the central server, the last call from its main routine should be tcaModuleListen(), which is essentially a loop that waits for a message from the central server, invokes the appropriate message handler, and then waits for another message. Alternatively, a module can call tcaHandleMessage(), which waits for a message, delivers it to a handler, waits for the handler to finish, and returns. You can specify how long tcaHandleMessage() should wait for a message to arrive before returning. Use of tcaHandleMessage enables you to do other tasks, such as monitoring the status of the robot hardware or attending to user inputs, in between handling messages

If a connection is broken, or if a module exits, the central server deletes the module registration information and continues as if the module were never registered. However, messages sent by other modules that would have been handled by that module will be queued, pending reconnection of the module (or another module that handles the same message). In this way, modules can be taken down and restarted without affecting the running of the system.

The following items are specifications and descriptions of TCA routines used in a module's initialization.

Open a connection to the central server.

void
tcaConnectModule(*moduleName*, *machineName*)
char **moduleName*, **machineName*;

Connects and initializes the module to the central server. It also verifies that the TCA library used to build the module has the same TCA major version number as the central server. If it doesn't, the module prints a message indicating the problem and then exits. If there is only a minor version difference between the module and the server, the module displays a warning, but it continues to run.

The string *moduleName* is used for logging and other display purposes. To identify the module, the central server relies on a low-level

Sample main() routine for a module that listens for service requests

The following abbreviated C code is a sketch of a main routine in a module that listens for messages and delivers them to low-level routines called handlers. See page 49 for real code examples.

```
/* Listener Module XYZ */
void main (void)
{
    /* Connect to the server */
    tcaConnectModule( "XYZModule",
            tcaServerMachine() );

    /* (Tell the server about data format names that you
            will use when you register messages.) */
    tcaRegisterNamedFormatter(parameter list…);
    …
    /* (Register messages that will be serviced by this module) */
    tcaRegisterQueryMessage(parameter list…);
    …
    tcaRegisterCommandMessage(parameter list…);
    …
    tcaRegisterInformMessage(parameter list…);
    …
    (registration for other types of messages)

    …
    /* (Register each routine that will perform appropriate
            action for a given message.) */
    tcaRegisterHandler(parameter list…);
    …
    /* (If we're just starting our system, wait until the other
            modules have registered their formats, messages,
            and handlers.) */
    tcaWaitUntilReady();
    /* (When the previous call returns, the next call will begin
             listening for messages and delivering them to
            the appropriate handler procedures.) */
    tcaModuleListen();
}

/* The rest of this module would comprise the handlers
    you defined, plus any supplementary routines needed
    by the handlers. */
```

communication ID, not on the name. Thus, it is possible to give the same name to more than one module.

The *machineName* is the Internet domain name of the host machine on which the central server is running. The routine tcaServerMachine() returns a value suitable for the *machineName* argument: Its description appears next.

Note: It is possible to have multiple modules of the same name on the same machine running at the same time. Whichever module is not busy will receive the request to handle a message. This could be a useful feature. For example, if two identical planning requests were made, it would be efficient to let each of two modules service the requests in parallel. Multiple modules can, however, cause conflicts. If you run two identical controller modules, for example, they might attempt conflicting control of the same hardware.

Get the name of the server machine.

char *
tcaServerMachine()

Returns the string value of the environment variable CENTRALHOST, or NULL if the variable is undefined. This procedure is typically used as the second argument to tcaConnectModule():

```
tcaConnectModule(“Controller”, tcaServerMachine() )
```

The string returned by this routine will typically be an internet domain name, e.g. “audrey.learning.cs.cmu.edu”; if more than one central server exists on a machine, then the string should also contain a colon and a port number, e.g. “audrey.learning.cs.cmu.edu:1621” to specify the appropriate server. See the box on page 5 for details.

Pause module until all other modules have finished registering messages and handlers.

TCA_RETURN_VALUE_TYPE
tcaWaitUntilReady()

Waits until central server signals that it is ready to process messages.

When you start the central server, you must specify the number of modules (*exp-mods*) that will be connecting to the system. (see page 5 for details.) Each module should contain a tcaWaitUntilReady() call after the last registration call so that all modules will have registered their handlers and messages before any module starts sending messages to the central server. This precludes sending a message before it has been defined or before a handler has been specified for it.

The central server counts calls to tcaWaitUntilReady() until the specified number is reached. At that point, all tcaWaitUntilReady() calls return, and the central server is free to forward messages. Note that *exp-mods* is just the *minimum* number expected: additional modules may connect at any time. Further calls to tcaWaitUntilReady() will return immediately.

> Many of the TCA procedures return values of type TCA_RETURN_VALUE_TYPE. Unless specified otherwise, the only value that will be returned by such procedures is Success.

Close module.

void
tcaClose()

Closes the module’s connection to the central server.

When a connection is closed for any reason (call to tcaClose(), crash of module, etc.), the central server deletes the module’s registration information and continues as if the module had never been never registered. However, if a message is sent to that module, it is queued until that module reconnects, or until another module that handles the same message connects.

Registering messages

To register a message with the TCA central server, you must specify:

- the name of the message;
- the format of the data you will send with the message;
- the format of the data the message’s handler will send in reply to the message (if a reply is expected).

Message names should be unique. If a module registers a name that has already been registered, the central server will issue a warning message, and the new specification will supersede the old one. Typically, each message is registered in the module that handles it.

For example, *query* messages in TCA pass data to a handler, and they expect a reply. (*Query* is one of several different classes of messages, each of which will be discussed in detail later.) Here is the procedure specification for the tcaRegisterQueryMessage() routine:

Register a query message.

TCA_RETURN_VALUE_TYPE
tcaRegisterQueryMessage(*msgName*, *msgForm*, *replyForm*)
char **msgName*;
char **msgForm*, **replyForm*;

Registers *msgName* as the name of a query message that sends a data structure specified by *msgForm*. The message handler will reply by sending a data structure specified by *replyForm*.

The data structure specifiers used for *msgForm* and *replyForm* are called “format strings.” See the next section "Registering message data formats" for more on format strings. The *msgForm* string may be NULL (NIL in LISP) to indicate that no data is to be sent with the query (the *replyForm* must not be NULL).

Registering message data formats

This section explains how to describe data structures that you will pass in your messages. The module that sends a message and the module that receives it must share a common data structure definition to enable data transfer. In addition, to allow TCA's communication routines to pass data *transparently* between processes, you must specify the data formats used by each message. You provide such a structural specification (called a "format string") in parameters to message registration routines. Once this is done, TCA can know how to convert the data structure to a byte stream and how to reconstruct it in the receiving module.

Suppose that you need to register a query message called "GetData." It passes a single integer and expects a character in response. You would use the following call to define it:

```
tcaRegisterQueryMessage("GetData", "int", "char");
```

Generally, however, you'll need to pass more complicated data structures. Suppose that your message must pass a data structure containing an integer, a character string, and another integer, and it will expect two characters and an integer in reply. This call would register the message:

```
tcaRegisterQueryMessage("GetData", "{int, string, int}",
    "{char, char, int}")
```

Rather than specifying data formats directly in message registration calls, we recommend that you first define a data type, define a TCA format specifier for that type, and then use the TCA form in the message registration call:

```
typedef struct {
    int x;
    char *y;
    int z;
} DATA1_TYPE;

#define DATA1_FORM "{int, string, int}"

typedef struct {
    char x;
    char y;
    int z;
} DATA2_TYPE;

#define DATA2_FORM "{char, char, int}"

#define GET_DATA_QUERY "GetData"

tcaRegisterQueryMessage(GET_DATA_QUERY,
    DATA1_FORM, DATA2_FORM);
```

By keeping the type definitions close to the TCA format specifiers, you ensure that if you need to make changes to a type, you can quickly locate and change the corresponding TCA definitions. We also recommend that you define all message names as macros to avoid the possibility of misspelling message names.

TCA formats for primitive data types

The previous example used the form "string" to stand for a list of characters. TCA also provides names for other data primitives; some of these names coincide with standard C language types, while others do not. Here is a complete list of primitives:

- byte: any 8-bit piece of information;
- char: an 8-bit piece of information, probably an ASCII code;
- short: any 16-bit piece of information;
- int: 32 bits of information;
- float: 64 bits of information;
- double: 128 bits of information;
- Boolean: information that takes on one of two values: TRUE or FALSE. In C, 0 is FALSE, while 1 is TRUE. In Lisp, NIL is FALSE, while T is TRUE.
- string: A list of characters—in C, this list is terminated by NULL ('\0');
- TCA_REF_PTR: a reference to a particular instance of a message (this is rarely used by application modules, but is used by TCA itself).

See page 50 for sample code that demonstrates the use of these data types.

TCA formats for composite data types

Composite data formats are aggregates of other data types. The supported composites include:

Structures

To describe a C language "struct" to TCA, surround the component data type names with braces, and place commas between them.

```
typedef struct {
    int x;
    char *str;
    int y;
} DATA_TYPE;

#define DATA_FORM "{int, string, int}"

#define USE_DATA_COMMAND "UseData"

tcaRegisterCommandMessage(GET_DATA_QUERY,
    DATA_FORM);
```

Structures can be nested. For example, a pair of DATA_TYPE components could be specified as follows:
"{ {int, string, int}, {int, string, int} }"

See page 52 for an example of structure usage in TCA programs.

Fixed-length and Variable-length Arrays

To describe a fixed-length array, use the following form:

[*data_type*: *n*]

data_type is the base type of the array, and *n* is the array dimension. If the array is multi-dimensional, separate the dimension numbers by commas, as in the following example:

```
typedef int array[17][42] DATA_TYPE;
```

```
#define DATA_FORM "[int:17, 42]"
```

Variable-length arrays are specified as part of a larger structure, which must contain "int" elements specifying dimensions. The notation

```
<char: 1,2,3>
```

indicates that a three dimensional array is contained in a larger data structure whose 1st, 2nd, and 3rd elements specify the dimensions of the array.

For example, a two-dimensional variable array of integers can be specified by placing it inside of the following structure:

```
typedef struct {
    int dimension1;
    int dimension2;
    int variableArray[];
} VARIABLE_ARRAY_TYPE;
```

The appropriate TCA format string would be:

```
#define VARIABLE_ARRAY_FORM "{int, int, <int: 1, 2>}"
```

See page page 54 for an example of array usage in TCA programs.

Pointers, linked lists, recursive data structures

Pointers are denoted by an asterisk followed by a data format name. If the pointer value is NULL (or NIL in Lisp) no data is sent. Otherwise the data is sent and the receiving end creates a pointer to the data. Note that only the data is passed, not the actual pointers, so that structures that share structure or point to themselves (cyclic or doubly linked lists) will not be correctly reconstructed.

```
typedef struct {
    int x, *pointerToX;
} POINTER_EXAMPLE_TYPE;

#define POINTER_EXAMPLE_FORM "{int, *int}"
```

The "self pointer" notation, *!, is used in defining linked (or recursive) data formats. TCA will translate linked data structures into a linear form before sending and then recreate the linked form in the receiving module. TCA routines assume that the end of any linked list is designated by a NULL (or NIL) pointer value. Therefore it is important that all linked data structures be NULL terminated so that the data translation routines work correctly.

```
typedef struct _LIST {
    int x;
    struct _LIST *next;
} LIST_TYPE;

#define LIST_TYPE_FORMAT "{int, *!}"
```

See page 56 for an example of pointer usage in TCA programs.

Registering your own data format names

You can define composite data formats by using the procedure tcaRegisterNamedFormatter(), which takes two arguments: a name string, and a format string. TCA replaces the name string with the format string when it parses a data format specification. For modularity reasons, it is recommended that you register a named formatter for each type definition (typedef).

Names assigned by tcaRegisterNamedFormatter() in one module may be used by other modules in registering messages: It is only necessary to register a form once. Once defined, a format is available to all modules. Format names are not expanded until a module makes its tcaWaitUntilReady() call.

See page page 57 for an example of user-defined format usage in TCA programs.

Assign a name to a format string

void
tcaRegisterNamedFormatter(*name*, *formatString*)
char **name*, **formatString*;

Allows the user to define a name to use in place of a format description. This saves typing and allows for more descriptive names in format definitions. It also increases system modularity.

```
typedef struct {
    float x, y, z;
 } POINT_TYPE;

#define POINT_FORM "{float, float, float}"

tcaRegisterNamedFormatter("point", POINT_FORM);
```

After "point" has been defined, it may be used in subsequent TCA format definitions:

```
typedef struct {
    POINT_TYPE start, end;
 } SEGMENT_TYPE;

#define SEGMENT_FORM "{ point, point }"

tcaRegisterNamedFormatter("segment",
    SEGMENT_FORM);

tcaRegisterQueryMessage("FindStart", "segment", "point");
```

Number of bytes: an alternative format specifier (C users only)

In addition to specifying data as a list of defined primitives, one can also give an integer number specifying the number of bytes to send. In the example below, each format would be equivalent:

```
typedef struct {
    int x;
    int y;
    char *string;
} SAMPLE_TYPE;

#define SAMPLE_DATA_FORMAT_1 "{int, int, string}"

#define SAMPLE_DATA_FORMAT_2 "{4, 4, string}"

#define SAMPLE_DATA_FORMAT_3 "{8, string}"
```

Using this feature could introduce implementation dependencies into your code. Use it only if you have no alternative.

This option is *not* available in Lisp.

Assign a name to a data structure that uses a given number of bytes (available only in C)

void
tcaRegisterLengthFormatter(*name*, *length*)
char **name*;
int *length*;

Allows the string *name* to specify a data structure whose size (in bytes) is given by *length.* For example, to let the name "triple" refer to a 96-byte data structure, use the following call:

```
tcaRegisterLengthFormatter("triple", 96);
```

After you make this call, you may use "triple" as you would use other data primitives, such as "double." We do not recommend using this routine, but it is provided for completeness.

Handling messages

When a module sends a message to the central server, it makes a request for a particular service. One of your modules must contain a procedure, called a *handler*, that provides the service. Modules must inform the central server about their message handlers by using tcaRegisterHandler(). You must specify the name of the message to be handled, a name for the handler, and a pointer to the procedure that will handle the message.

When the central server receives a message, it consults its routing information and forwards the message to the module that has registered a handler for the message. The module is likely to be executing either the tcaHandleMessage() procedure or the tcaModuleListen() procedure (see below). Either of these will accept the message, and then automatically forward it—along with its data—to the correct handler procedure.

Each of your handlers must accept exactly two parameters. The first is a pointer to a message reference (of type TCA_REF_PTR), which will be used to respond to the message. The second is a pointer to the data that is sent with the message. The data is in the format that you specified when you registered the message with the central server. The handler can make use of the data as it sees fit to process the message request.

Handlers tell the central server they have finished handling a message by calling tcaReply(), tcaSuccess(), or tcaFailure()—depending on the class of message and the completion status of the request. Generally, this is the last call made by the handler, although it does not have to be. (If, for example, the last thing a handler must do is provide a graphical display of some result and this display is time-consuming and does not affect subsequent tasks, it might be more efficient to send a reply *before* performing the display function.)

The central server maintains information on which handlers are currently processing messages. By default, a module can execute only a single handler procedure at a time, and messages are executed in FIFO order. The "Resource Management" chapter describes how to override these defaults.

Describe a message handler to the central server.

TCA_RETURN_VALUE_TYPE
tcaRegisterHandler(*msgName*, *hndName*, *hndProc*)
char **msgName*, **hndName*;
void (**hndProc*)();

Defines a handler name *hndName* for the message handler procedure *hndProc*, which handles the message *msgName*. The procedure *hndProc*(), a procedure with no return value (void), will be called by the TCA communication routines in response to a request by the central server to handle the message *msgName*. The procedure *hndProc*() will be called with two parameters: *msgRef* and *data*. *msgRef*, of type TCA_REF_PTR, is an internal reference to the origin of the message. It is used primarily as an argument to other task control routines that *hndProc*() may call, such as tcaReply(). *data* is a pointer to user-defined data. Its format is defined by the *msgForm* that you provided when you registered the message.

For samples of message handlers, see the examples starting on page 49.

Remove a message handler from the system.

TCA_RETURN_VALUE_TYPE
tcaDeregisterHandler(*msgName*, *hndName*)
char **msgName*, **hndName*;

Tells TCA that the handler *hndName* is no longer applicable to processing message *msgName*. *msgName* and *hndName* should be identical to that used in the tcaRegisterHandler() call.

This call can be used anytime and in any place in your module. It affects all messages sent after the deregistration request is received by the central server. If no handler is registered when a message is sent, TCA will queue the message until a new handler is registered and then forward the message to that handler.

While this call can be used for any message class, it is particularly useful for broadcast and multi-query class messages (described later in this chapter).

Handle a single message from the central server and return.

TCA_RETURN_VALUE_TYPE
tcaHandleMessage(*waitValue*)
long *waitValue*;

Accepts a single message from the central server, invokes the appropriate handler procedure, and then returns when the handler completes. The argument *waitValue* is the number of seconds to wait for an incoming message from the central server. The defined value WAITFOREVER can be used to cause tcaHandleMessage() to wait indefinitely until a message request is received. tcaHandleMessage() will return TimeOut if it fails to handle a message within the number of seconds specified by *waitValue*. tcaHandleMessage() returns Success if it successfully handled a single message.

Loop for processing central server message requests.

TCA_RETURN_VALUE_TYPE
tcaModuleListen()

Waits for the central server to forward a message, invokes the appropriate handler procedure, and then waits for additional messages. This procedure loops indefinitely—it never returns.

Multiple Handlers for a Single Message

If multiple handlers are registered for the same message, TCA will randomly select the first handler it finds in a resource that is available. (Exception handlers are a special case. See "Task trees and the context of exception handling" on page 41 for more information.)

Multiple Messages for a Single Handler

It is possible that the services required by several different messages have enough in common that a single procedure could service them. You can register the same handler for different messages, even if they have different data formats. TCA will pass the correct data.

Suppose, for example, that you have a procedure called ABHandler() that will service messages called “MessageA” and “MessageB,” and you make the following two registrations:

```
tcaRegisterHandler(“MessageA”, “ABHandler”, ABHandler);
tcaRegisterHandler(“MessageB”, “ABHandler”, ABHandler);
```

When you send MessageA, the central server will pass the message and its data description to ABHandler(). When you send MessageB, the central server will pass MessageB and *its* data description to ABHandler(). Thus, ABHandler() will be able to decode the data for each message, even though the formats may be different.

At some point in the procedure, you may need to find out which message was delivered to the handler so that the procedure can take appropriate action:

if message = “MessageA” do X
if message = “MessageB” do Y

You can get the message’s name by passing the handler’s *ref* parameter to tcaReferenceName().

Sending messages

To send a message, you only need to tell TCA the name of the message, the data to be sent with the message, and (optionally) a data structure that will be filled with the response to the message. You do not have to know which module handles the message—TCA routes the message appropriately based on the message registration calls. You do not have to know how to package the data to be sent—TCA knows how to do that as well, based on the format string registered with the message. In this way, you can easily change the format of the data structures sent or which module will handle the message (e.g., a simulator or the real vehicle controller) without having to change anything in the sending (or receiving) modules.

Note that TCA handles message routing correctly, even if a module sends a message to itself. While this could also be accomplished with a direct procedure call, sending a message is more general because a) you can change which module handles the message without having to change the code that sends the message, and b) TCA can handle situations in which you want to sequence or synchronize the handling of the message with respect to other messages in the system.

TCA messages can be *blocking* or *non-blocking* (some message classes can be either, but most are one or the other). Blocking messages pause the sending module until the service requested has been completed. Blocking message calls return status and/or information that indicate how the message was handled. This is, of course, the way procedure calls ordinarily work; when your code calls a procedure, it generally does not go to the next instruction until the call returns.

Non-blocking calls return control to the module immediately after sending the message. There is no guarantee that the message has been handled by the time the call returns. Non-blocking messages are useful if you want modules to be processing concurrently—the module that sends a non-blocking message is free to continue computing (including sending additional messages) while the message is being handled by another module. TCA ensures that non-blocking messages are sequenced appropriately (see chapters on “Resource Management” and “Task Management”). Most of the TCA message calls are non-blocking, enabling modules to make efficient use of computational resources.

Sometimes it is necessary for a module to check on the status of a non-blocking message. This can be done using the call tcaReferenceStatus(), described in the “Task Management” chapter.

The procedure you use to send a message depends on the class of the message. In the rest of this section, we will describe several of the classes of messages provided by TCA: queries, commands, inform, broadcast and multi-query class messages. Goal, monitor and exception class messages will be discussed in subsequent chapters.

> When making procedure calls, particularly for sending messages, be careful to provide *pointers* to data structures rather than the structures themselves. Trying to pass structures where pointers are required is a very common problem in TCA programs. In particular, if the data is itself implemented as a pointer (e.g. arrays and strings), you must pass a pointer to this pointer.

Query messages

A common service in distributed systems is that one module requires information from another module. Typically, the module queries for some specific information (such as the value of a sensor) and receives a reply. This type of service is provided by the procedure tcaQuery(). tcaQuery() is a *blocking* call.

Suppose that one of your modules were to make the following calls, where print_data() is a routine you have defined:

```
status = tcaQuery(“sample_query”, (void *)&q_data,
    (void *)&r_data);
if (status != NullReply) print_data(r_data);
```

The “sample_query” message would be sent to the central server, along with the query data, “q_data.” Your module would then wait for the following steps:

- the central server forwards the message to the appropriate module, which executes a handler;
- the handler processes the query data and sends the reply data;
- the central server forwards the reply data to the module that sent the query.

Until “sample_query” has been processed and the reply data is available in “r_data,” the calling module will not proceed to the next line in the program.

Send a query message and wait for a reply.

TCA_RETURN_VALUE_TYPE
tcaQuery(*queryName*, *queryData*, *replyData*)
char **queryName*;
*void *queryData*;
*void *replyData*;

Sends a message to the central server, which then routes it to the module that handles that type of message. This call waits (blocks) until the message is handled and a reply is received. The parameter *queryData* is a pointer to a structure that contains arguments for the query. The parameter *replyData* is a pointer to a structure into which the handler will place data in response to the query message. (Note: *replyData* must point to a structure that has already been allocated). The data types of *queryData* and *replyData* must correspond to the format strings indicated in the message’s corresponding tcaRegisterQueryMessage() call.)

Typically, the return value of a call to tcaQuery() will be Success. As a special case, if the message handler was unable to process the query, the return value will be NullReply (see page 13). The value of *replyData* is undefined if the tcaQuery() returns the value NullReply.

See page 58 for an example of blocking query usage in TCA programs.

Suppose that you know you will eventually need some information from your system, but you also know that computing or gathering the information will take a substantial amount of time. If your module has other tasks to perform, an efficient approach might be this:

- Request the information, but don’t wait for it;
- Proceed to other tasks;
- When you’ve finished your other tasks, retrieve the information you requested earlier.

In the ideal case, the process responsible for your first request will have finished its work by the time you are ready for it; retrieving the information would be nearly instantaneous.

What you need is a *non-blocking* form of tcaQuery(). Your module can then proceed to other things while the server forwards the message to the appropriate query handler, which should process the message and send a reply to the central server. You can retrieve the reply at your convenience (if it is ready) or wait for it (if processing hasn’t finished). Here is an example of such an exchange:

```
void some_routine(void)
{
    TCA_REF_PTR queryRefA, queryRefB;
    QUERY_A_TYPE queryDataA;
    QUERY_B_TYPE queryDataB;
    REPLY_A_TYPE replyDataA;
    REPLY_B_TYPE replyDataB;
    …
    /* both QueryA and QueryB take a long time to answer,
         so we send them now. Both calls return immediately. */
```

```
tcaQuerySend("LongMessageA", (void *)&queryDataA,
        &queryRefA);
tcaQuerySend("LongMessageB", (void *)&queryDataB,
        &queryRefB);
…
… execute other commands in the meantime …
…
/* At this point, we need the data from Query B, so we'll
        retrieve it, waiting for it if necessary */
tcaQueryReceive(queryRefB, (void *)&replyDataB);
…
/* now we need the data from QueryA, so we'll
        retrieve it, waiting if necessary */
tcaQueryReceive(queryRefA, (void *)&replyDataA);
…
}
```

Remember that the handlers for the two queries could be in different processes, possibly on different processors. For this reason, non-blocking requests allow for concurrent operation.

The procedure that sends a non-blocking query message is described next; immediately after it is the procedure for retrieving the reply:

Send a non-blocking query message

TCA_RETURN_VALUE_TYPE
tcaQuerySend(*name*, *queryData*, *ref*)
char **name*;
void **queryData*;
TCA_REF_PTR **ref*;

Sends a non-blocking query message. Given the *name* and *queryData*, this routine will assign a value to *ref* that you will use when you call tcaQueryReceive(). Remember, you must pass a *pointer* to a TCA_REF_PTR variable (see example on the preceding page).

Wait for and receive reply to non-blocking query

TCA_RETURN_VALUE
tcaQueryReceive(*ref*, *replyData*)
TCA_REF_PTR *ref*;
void **replyData*;

Waits for a reply to a tcaQuerySend(). The *ref* passed must be the same as the one obtained from the tcaQuerySend() function, but the send's and receive's do not have to be in the same order. That is, you can send query1, send query2, receive query2, receive query1. TCA makes sure the right replies get paired with the right receives. (See the example on the preceding page.)

Typically, the return value of a call to tcaQuery() will be Success. As a special case, if the message handler was unable to process the query, the return value will be NullReply (see page 13). The value of *replyData* is undefined if the tcaQuery() returns the value NullReply.

See page 59 for an example of non-blocking query usage in TCA programs.

Replying to query messages

When a handler is finished processing a message, it must notify the central server that the message has been handled. The following two routines may be used by handlers to reply to query messages. Generally, one of these procedures would be the last call in a handler.

(Routines for responding to other classes of messages will be presented when the other classes are introduced.)

Respond to a query with some information

TCA_RETURN_VALUE_TYPE
tcaReply(*ref*, *replyData*)
TCA_REF_PTR *ref*;
void **replyData*;

When a message handler is invoked, the first argument will be a reference pointer: use this value for *ref*. The parameter *replyData* is a pointer to the data to be returned to the issuer of the query. It should have the same format as that defined as the *replyForm* used when the message was registered (see page 7).

See page 58 for an example.

Respond to a query by sending no information

TCA_RETURN_VALUE_TYPE
tcaNullReply(*ref*)
TCA_REF_PTR *ref*;

Used to indicate that the handler could not deal with the query for some reason (e.g., perhaps data passed to the handler was out of range). The call tcaNullReply() will reply to a query of message *ref* by sending no data and the return value of the tcaQuery() call will be NullReply. (If a handler responds with tcaReply(), the return value of the tcaQuery() call will be Success.) The routine that issued the query is responsible for handling return values of NullReply.

TCA issues a warning if a handler returns without having called tcaReply() (or tcaSuccess() or tcaFailure() for command and goal class messages). Typically this warning should be heeded, because deadlock can occur if the central server is not informed that a message handler has finished processing.

Deferring responses to a message

There are times that you might want a handler to return without replying/responding to a message, deferring the response until later. For example, suppose your module had both a TCA and an X interface. Your main program loop might look like:

```
while (TRUE) {
    /* Handle one message; do not wait if there are no messages */
    tcaHandleMessage(0);
    handleXEvent();
}
```

How could you use the X interface to acquire information from the user to respond to a query message? First, you could set up a pop-up window, save the *ref* pointer (the first argument to the handler) in a global variable, and then return from the handler. This would drop you into the main loop (returning from the tcaHandleMessage() call), and enter the code to handle X events. Eventually, when the user enters the appropriate information, the X callback procedure would call tcaReply(), using the saved *ref* pointer as its first argument.

There are two problems with this scenario. First, when the handler returns, TCA will print a warning message because it thinks that you have forgotten to reply to the message. Second, and more importantly, TCA automatically frees the message *ref* pointer when a handler returns. Thus, the saved reference would point to garbage.

The solution to both problems is to tell TCA that you will be deferring responses to messages. You do this by calling tcaEnableDistributedResponses(). This should be done just once, at the time that you register messages. It then applies to all messages within that module. This does not mean that all your messages must have deferred responses, only that it is permissible for any message handler in that module.

tcaEnableDistributedResponses() can also be applied to other message classes that send a response (i.e., tcaSuccess() and tcaFailure()). For example, a command handler in a controller module can start an actuator moving and then return; later, when the actuator position has been reached, another part of the code can respond with tcaSuccess().

> When distributed responses are enabled, TCA frees the *ref* pointer whenever tcaReply(), tcaSuccess(), or tcaFailure() are called. Thus, one must not use the *ref* after such calls.

Permit deferred responses to messages.

void
tcaEnableDistributedResponses(*void*)

Enable message handlers in a module to return without issuing a tcaReply(), tcaSuccess() or tcaFailure() call in response to a message. The responses can then be made subsequently, by other parts of the code. One side-effect of calling tcaEnableDistributedResponses() is that TCA frees the *ref* pointer used in the reply calls (normally, this is done automatically when the handler returns).

Should be called just once, when messages are registered. Applies to all message handlers in a module.

Prohibit deferred responses to messages.

void
tcaDisableDistributedResponses(*void*)

Enforces the constraint that all message handlers must issue a reply (if one is expected for the message class). This is the default behavior. It is unclear if this function is actually useful, but it is included for completeness and symmetry.

Command messages

To request a physical action from your robot, send a *command* message. Procedures that send command messages take as arguments a message name and a pointer to a data structure that parameterizes the action. Both blocking and non-blocking command requests are available.

The order and time of execution depends on available resources and the temporal constraints placed on the command. By default, commands are executed in the order they are received, provided the physical resource they use is available. The default can be changed by using tcaExecuteCommandWithConstraints(), which will be discussed when we present temporal constraints in the chapter “Task Management.”

Command message handlers must respond by calling either tcaFailure() or tcaSuccess(). The “Success” or “Failure” messages are routed to the central server. TCA will log a warning if a command message handler returns without sending an appropriate response.

Register a command message.

TCA_RETURN_VALUE_TYPE
tcaRegisterCommandMessage(*msgName*, *msgForm*)
char **msgName*;
char **msgForm*;

Registers the string *msgName* as a valid command message name. The string, *msgForm* is a data format string. See "Registering message data formats" on page 8 for more on format strings. *msgForm* may be NULL (NIL in LISP) indicating that no data is to be sent with the command.

Execute a command and wait for completion (blocking request).

TCA_RETURN_VALUE_TYPE
tcaWaitForCommand(*commandName*, *data*)
char **commandName*;
void **data*;

Sends the command message *commandName* and waits until the associated action is attempted. If the handler for the message replies with a tcaSuccess(), the return value is Success; otherwise, the return value is Failure. The type of *data* depends on the particular message, but must correspond to the format string used in the associated tcaRegisterCommandMessage() call. The pointer may be NULL (NIL in LISP), indicating that no data is to be sent.

See page 60 for an example of command usage in TCA programs.

Execute a command (non-blocking request).

TCA_RETURN_VALUE_TYPE
tcaExecuteCommand(*commandName*, *data*)
char **commandName*;
void **data*;

Sends the command class message *commandName* and returns control immediately. The command will be forwarded to its handler, and *data* will be used as the handler's second parameter value. Returns a status value; currently Success is the only defined value. The type of *data* depends on the particular message, but must correspond to the format string used in the tcaRegisterCommandMessage() call. The pointer may be NULL (NIL in LISP), indicating that no data is to be sent.

> Because this routine is non-blocking, Success means only that the request was successfully sent to the central server. It does not mean that the request has been handled. Although the central server knows whether the command succeeded or failed, the sender is not notified. Separate exception handling facilities are provided to deal with command failures. (See "Exception handlers" on page 39.)

Responding to command messages

Send "Success" in response to a command.

TCA_RETURN_VALUE_TYPE
tcaSuccess(*ref*)
TCA_REF_PTR *ref*;

Indicates that the command message was successfully executed. This procedure should be the last call made by a handler before it returns. The parameter *ref* will be the first argument passed to the command message handler.

Send "Failure" in response to a command

TCA_RETURN_VALUE_TYPE
tcaFailure(*ref*, *description*, *failureData*)
TCA_REF_PTR *ref*;
char **description*;
void **failureData*;

Indicates an execution-time (command) failure. For example, suppose that your robot control system contained a "Move" command. message. If you sent a "Move" message and the handler could not execute the command—perhaps because the robot would collide with an object—the handler would call tcaFailure(), e.g.,

```
tcaFailure(ref, "Move failed! Potential collision",
        (void *)&distanceToObject)
```

The parameter *ref* is either a message reference, if tcaFailure() is being called from within a handler, or NULL if it is called from outside a handler (at top level).

The *description* string can be one of two things: a short error message, as in the example above, or the name of an exception message. If the string is not the name of an exception message, the failure is simply logged by the central server, and processing continues. Otherwise, an exception message is sent, along with the data in *failureData*. The appropriate exception handler will attempt some recovery action.

Inform messages

Suppose that one of your modules must provide information to another module—the desired speed of your robot, for example. You don't want to have an action performed, nor do you require an answer to your message. In this case, a message of the *inform* class is appropriate.

To send an inform message, use tcaInform(). It takes two parameters: the message name, and a pointer to the information data. It does not wait for a reply, so it returns immediately. Moreover, it does not *expect* a reply. Handlers that you define for inform messages should not issue responses.

Inform messages are handled in the order they are received, subject to resource availability.

> In previous versions of TCA, inform class messages were called *constraint* messages. The name was changed to be more descriptive of usage. While, for upward compatibility, the old constraint class procedure calls are still supported, users are encouraged to switch to using tcaInform() in their modules.

Register an inform message.

TCA_RETURN_VALUE_TYPE
tcaRegisterInformMessage(*msgName*, *msgForm*)
char **msgName*, **msgForm*;

Registers the string *msgName* as a valid inform message name. The string, *msgForm* is a data format string. See "Registering message data formats" on page 8 for more on format strings. *msgForm* may be NULL (NIL in LISP) indicating that no data is to be sent with the message.

Register a constraint message.

TCA_RETURN_VALUE_TYPE
tcaRegisterConstraintMessage(*msgName*, *msgForm*)
char **msgName*, **msgForm*;

Identical to tcaRegisterInformMessage(). Supported only for upward compatibility with older versions of TCA.

Send information to a module.

TCA_RETURN_VALUE_TYPE
tcaInform(*informName*, *data*)
char *informName;
void *data;

Sends the information-only message *informName* with the information in *data*. Returns a status value; currently Success is the only defined value. The data type depends on the particular message, but must correspond to the format string used in the associated tcaRegisterInformMessage() call. The pointer may be NULL (NIL in LISP), indicating that no data is to be sent.

Inform messages do not expect replies, so inform message handlers should not send them.

Send information to a module.

TCA_RETURN_VALUE_TYPE
tcaAddConstraint(*constraintName*, *data*)
char *constraintName;
void *data;

Identical to tcaRegisterInformMessage(). Supported only for upward compatibility with older versions of TCA.

Broadcast Messages

The message classes described so far have all been point-to-point: one sender and one receiver. Sometimes, you need to broadcast a single message to any number of receivers. For example, you might want to inform modules of the updated position of the robot, instead of having to have them query for the position each time.

TCA provides broadcast class messages for this purpose. Any number of handlers can be registered to handle a broadcast message (including multiple handlers in the same module). When the broadcast message is sent, TCA forwards a copy of the message data to each handler that registered for that message. If no handler has been registered for the broadcast, TCA just discards the message data. Broadcast messages are like inform class messages: the handlers for broadcast messages should not issue a response. Broadcast messages are handled in the order they are sent.

Note that for other message classes, it is customary to register the message and message handler together. For broadcast messages, it is customary to register the message in the module that issues the broadcast, and to register handlers in the receiving modules.

You can alter the set of messages receiving broadcasts by calling tcaRegisterHandler() and tcaDeregisterHandler() at various points and times in your system. Registering and deregistering broadcast handlers takes effect as soon as the central server receives notification.

Register a broadcast message.

TCA_RETURN_VALUE_TYPE
tcaRegisterBroadcastMessage(*msgName*, *msgForm*)
char **msgName, *msgForm*;

Registers the string *msgName* as a valid broadcast message name. The string, *msgForm* is a data format string. See "Registering message data formats" on page 8 for more on format strings. *msgForm* may be NULL (NIL in LISP) indicating that no data is to be sent with the broadcast.

Broadcast information to all interested modules.

TCA_RETURN_VALUE_TYPE
tcaBroadcast(*broadcastName*, *data*)
char **broadcastName*;
void **data*;

Sends the broadcast message *broadcastName* with the information in *data*. Returns a status value; currently Success is the only defined value. The data type depends on the particular message, but must correspond to the format string used in the associated tcaRegisterBroadcastMessage() call. The pointer may be NULL (NIL in LISP), indicating that no data is to be sent.

Multi-Query Messages

Suppose you had several ways to answer a query, but did not know in advance which method would give you the best, or fastest, reply. For example, suppose you could detect obstacles using sonar, laser or vision, but did not know which would work best in the current situation. You might want to send a message to all modules that can handle the query, and just wait for the first (or first several) responses.

Multi-query class messages are designed for this type of communication. Any number of handlers can register for the same multi-query message. You, in effect, send a query message to all of them simultaneously, and wait as the replies come in one at a time. You can either limit the number of replies you want (e.g., just the first two), or can get all of them. In the latter case, TCA will tell you when you have received all the replies.

Handlers for multi-query class messages behave just like query handlers: they receive two arguments (a TCA_REF_PTR and the query data) and must respond using tcaReply(). You can alter the set of messages receiving broadcasts by calling tcaRegisterHandler() and tcaDeregisterHandler() at various points and times in your system. Registering and deregistering broadcast handlers takes effect as soon as the central server receives notification.

Register a multi-query message.

TCA_RETURN_VALUE_TYPE
tcaRegisterMultiQueryMessage(*msgName*, *msgForm, replyForm*)
char **msgName*, **msgForm, *replyForm;*

Registers the string *msgName* as a valid broadcast message name. The strings*msgForm* and *replyForm* are data format strings. See "Registering message data formats" on page 8 for more on format strings. *msgForm* may be NULL (NIL in LISP) indicating that no data is to be sent with the query.

Send a multi-query message.

TCA_RETURN_VALUE_TYPE
tcaMultiQuery(*msgName*, *queryData, maxReplies, refPtr*)
char **msgName*;
void **queryData*;
int *maxReplies*;
TCA_REF_PTR **refPtr;*

Send a multi-query class message to all handlers registered for that message. *msgName* is the name of a multi-query message and *queryData* is a pointer to the data to be sent. The type of *queryData* depends on the particular message. *queryData* may be NULL (NIL in LISP) if no data needs to be sent.

TCA will send at most *maxReplies* responses (the actual number of replies may be fewer, depending on how many handlers are currently registered at the time of the multi-query). To get all the responses, set *maxReplies* to a very large number. If more than *maxReplies* handlers respond to the message, only the first *maxReplies* will be forwarded; the remaining replies will be discarded by the central server.

refPtr is a *pointer* to a variable of type TCA_REF_PTR. It is filled in with a reference to the query. It is used in the associated tcaMultiReceive() call (see below).

Receive response from a multi-query message.

TCA_RETURN_VALUE_TYPE
tcaMultiReceive(*refPtr, replyData, timeout*)
TCA_REF_PTR *refPtr;*
void **replyData*;
long *timeout*;

Receive one response from a multi-query message. The *refPtr* is filled in by the call to tcaMultiQuery() and the *replyData* is a pointer to a structure to be filled in with the next response.

timeout specifies how long to wait (in seconds) for the next response. The constant WAITFOREVER indicates that the call should not return until the next reply is received (or until no more response will be forthcoming). If *timeout* seconds pass before a reply is received, the return value of tcaMultiQuery() is TimeOut.

To receive all the responses to a multi-query, you should put the call to tcaMultiQuery() in a loop. The return value of NullReply indicates that no more replies will be forthcoming.

For example:

```
tcaMultiQuery("MultiQuery", &data, 3, &mqRefPtr);
for (;;;) {
    if (tcaMultiReceive(mqRefPtr, &replyData,
            WAITFOREVER) == NullReply) {
     break;
    } else {
    /* process reply here */
    }
}
```

Reclaiming data space (C users only)

Because C does not provide for automatic garbage collection, TCA provides several memory management routines.

When a message handler is invoked, memory is allocated for the data structures that are passed to the handler. When the handler returns, the storage allocated for the *ref* argument is freed, but the storage for the *data* argument is not.

Because *data* may comprise several pointer-based subelements, there is no trivial way to free the memory associated with it. For this reason, TCA provides tcaFreeData(), along with tcaFreeReply() for query handlers. Two other routines are also provided for flexibility.

For example:

```
void sampleMsgHandler(ref, data)
TCA_REF_PTR ref;
SAMPLE_TYPE *data;
{
    .
    .
    tcaQuery("query1", data, (void 80&r_data);
    .
    .
    tcaFreeReply("query1", (void *)&r_data);
    tcaFreeData(tcaReferenceName(ref), (void *)data);
    tcaSuccess(ref);
}
```

We recommend that you use tcaReferenceName() to get the name of the message currently being handled. This avoids misspellings and enables you to change the name of the message without having to change the handler code.

Free data created by TCA using the message format of a message.

void
tcaFreeData(*msgName*, *data*)
char **msgName*;
void **data*;

Recursively frees the data structure pointed to by *data* using the message format registered with the message, *msgName*.

The type of *data* depends on the message type.

Free reply data created by TCA using the reply format of a query message

void
tcaFreeReply(*msgName*, *replyData*)
char **msgName*;
void **replyData*;

Frees all of the allocated subelements (i.e., any pointer-based structures for which a malloc() call was required) of *replyData,* the data sent in response to the query message specified by *msgName*. Unlike tcaFreeData(), tcaFreeReply() does not free the top level structure of the data.

Register a user-level routine to attempt recovery from a memory allocation error

void
tcaRegisterFreeMemHnd(*func*, *retryCount*)
void (**func*)();
int *retryCount*;

TCA traps memory allocation errors. If TCA is unable to allocate the requested number of bytes to a malloc() call, the routine pointed to by *func* will be called and passed the number of bytes the TCA malloc() call failed on (an unsigned int). The function *func* should try to free enough memory so that the malloc() call will succeed. This will be repeated until the malloc() call succeeds or the function has been called the number of times specified by *retryCount*. If the malloc() call still fails or no free memory handler was registered then TCA simply exits, displaying the amount of memory requested.

Register a substitute program for the standard malloc() routine

void
tcaRegisterMallocHnd(*func*, *retryCount*)
void *(**func*)();
int *retryCount*;

TCA will normally call the system malloc() to process memory requests. If a module wishes to handle its own memory allocation, a substitute procedure for malloc() can be registered with the tcaRegisterMallocHnd() call. TCA will start allocating memory for its own use beginning with a call to tcaConnectModule(). Both tcaRegisterMallocHnd() and tcaRegisterFreeMemHnd() should be called before calling tcaConnectModule().

TCA will call the routine pointed to by *func*() and the routine will be passed (an unsigned int) the number of bytes of memory requested. If the routine returns NULL, it will be called again until the number of times it is called is equal to the *retryCount*. If NULL is still returned then the routine registered with tcaRegisterFreeMemHnd() is called. After each call to the freeing routine TCA will again call *func*() the number of times specified by the *retryCount* if the allocation has not been satisfied. If the allocation and freeing process still fails, then TCA simply exits, displaying the amount of memory requested.

Resource management

In a distributed system, multiple requests may be made to the same module simultaneously, so the system must schedule orderly access to modules. It is also likely that in a running system, more than one request is being serviced (by different modules) at any given time. Because not all robot functions are compatible, a system must have some way of managing conflicting access requests.

For example, if a vision module is acquiring an image, it might want to ensure that the robot does not move while it is scanning. You need some way to block any movement-related procedures until the scan is finished. If the handlers related to movement were grouped together in a unit, and you had control over messages being sent to that unit, then you would have the control you need. TCA provides such units; they are called *resources*.

A TCA *resource* is a group of message handlers contained in a single module (separate process), plus a message queue for the handlers. Each module is considered, by default, to be a single resource—a resource bearing the same name as the module itself and containing all of its handlers. For example, if you register a module called "W_module" containing handlers called "X," "Y," and "Z," TCA would automatically create a resource called "W_module," which would manage messages for the handlers "X," "Y," and "Z."

The messages being processed at any given time by a resource are in the resource's *attending set*, whose size has an upper bound called the *capacity* of the resource. The *pending set* of queued messages is potentially unlimited in size. The default resource that TCA creates for each module has a capacity of one, because under most operating systems, a process can only attend to one task at a time. Under operating systems with multi-threaded processes, such as VxWorks, you could specify a resource capacity greater than one.

When the central server receives a message, it selects an appropriate handler and then checks if the resource associated with the handler has available capacity. If the resource is available, the message is flagged as being "attended to," and the message is immediately forwarded to the module to be handled. If the resource is unavailable, the message is queued in the pending set until the resource is available. The pending set is processed in FIFO order.

A special case occurs when a module sends a query message whose handler is contained in the same module (i.e., sending a query to yourself). The module is sending a message, so it is "busy"; by definition, the module's resource should not be able to process the query until it is no longer busy. Thus, we have deadlock. To avoid this complication, TCA adds query messages to the attending set regardless of the resource's capacity. Then, while the module is blocked waiting for the reply, TCA invokes the appropriate query handler, receives the reply, routes the reply back to the module, and the tcaQuery() call completes, unblocking the module.

Multiple resources can be associated with a single module. You can create a new resource and place handlers from a given module into it. When you add a handler to a resource, you remove it from its former resource, because each handler can be in only one resource at a time. All handlers in a resource, however, must be registered within the same module.

For example, the robot controller module may have one resource to handler the motion commands, one resource to handle status queries, and a resource to handle an "emergency stop" command. Thus, the robot system can be moving and responding to requests simultaneously.

A module can completely disable a resource by *locking* it. A lock prevents *any* message from being sent to the handlers in that resource—including messages originating from the module that owns the lock—until the lock is explicitly removed. Alternately, a module can gain control over the attending set of a resource by *reserving* it. Only the reserving module can send messages to the resource until that module explicitly cancels the reservation.

Examples of resource usage are given in the chapter on "Example Programs."

Building a Resource

To create a new resource, register its name and specify its capacity by using the procedure tcaRegisterResource(). To move handlers out of one resource and into another, use tcaAddHndToResource().

A handler can be added only to a resource created by the module that registered the handler.

Create a resource.

void
tcaRegisterResource(*resName*, *capacity*)
char **resName*;
int *capacity*;

Creates a resource called *resName*, with an attending set size of *capacity*. If you specify a capacity less than 1, the central server will log a warning, and the capacity will be set to 1. Resource names are module-specific, so different modules can have resources by the same name.

If the resource was already registered (or created by default for a module) then the capacity of the resource will be changed to the new value. In this way, you can override the TCA default values.

Note that a module always has a default resource with the same name as the module.

Add a handler to a resource.

void
tcaAddHndToResource(*hndName*, *resName*)
char **hndName*;
char **resName*;

Adds the handler *hndName* to the resource *resName*. The module exits and the central server logs an error if *hndName* has not been registered by the module calling tcaAddHndToResource(), or if the module has not registered the resource *resName*.

Generally, this call is used to move a handler out of the default, TCA-created module resource into a resource that you have defined. If *hndName* is already a user-defined resource, the central server will log a warning message, because it is unusual to move a handler out of one user-defined resource into another.

This routine can be used to move a handler from one resource to another at any time. However, if a handler becomes part of a different resource, messages for the handler that are already queued, pending resource availability, will base their execution on the previous resource association.

Locking a resource

Lock a resource to prevent messages from being processed by that resource.

TCA_REF_PTR
tcaLockResource(*resName*)
char **resName*;

Prevents any module from using the resource *resName*. The call returns when the resource has been successfully locked: If the resource is not already reserved or locked and is available to handle messages, the lock request is processed immediately. Otherwise the request is queued and processed when the resource becomes available (i.e., when all the messages already in the attending set have been handled). Note that the lock request has priority over any message in the pending set of the resource.

Messages sent to a locked resource will be queued, and they will be processed only after the lock has been canceled.

The return value is a reference to the resource being locked; you will use this reference to unlock the resource.

Lock a resource in a specified module.

TCA_REF_PTR
tcaLockModResource(*modName*, *resName*)
char **modName*, **resName*;

Prevents any module from using the resource *resName* in the module *modName*. Resources in different modules are allowed to have the same name, so tcaLockModResource() is used in cases where the resource name *resName* is ambiguous. The module name, *modName*, is used to specify the desired resource exactly. In all other respects, this procedure is the same as tcaLockResource().

Remove a lock on a resource.

void
tcaUnlockResource(*ref*)
TCA_REF_PTR *ref*;

Removes the lock on the resource represented by *ref*, where *ref* is the value returned by tcaLockResource() or tcaLockModResource(). If *ref* is not a valid lock on a resource, the calling module will be terminated (and the central server will log an error)

> All locks must be explicitly cancelled. If a module exits (or crashes) without unlocking a resource, the lock will remain in effect indefinitely. The handler that locked the resource need not be the one to unlock it, but the resource must be unlocked before the module can be used again.

Reserving a Resource

All reservations must be explicitly cancelled. If a module exits (or crashes) without cancelling a resource reservation, the reservation will remain in effect indefinitely. The handler that reserved the resource need not be the one to cancel the reservation, but the reservation must be cancelled before the module can be used again.

Reserve the named resource.

TCA_REF_PTR
tcaReserveResource(*resName*)
char **resName*;

Reserves the resource *resName* for exclusive use by the calling module. Only messages sent by that module will be processed. Messages from other modules will be queued, and they will be processed only after the reservation has been canceled.

The call returns when the resource has been successfully reserved: If the resource is not already reserved or locked and is available to handle messages, the reservation request is processed immediately. Otherwise the request is queued and processed when the resource becomes available. Note that a reservation request has priority over other messages in the pending set of the resource.

If a module has already reserved a resource, the tcaReserveResource() call will generate a warning (in the log or at the terminal) and a special NULL reference is returned. This can be checked with tcaReferenceName() of *ref*—the reference name will be the null string.

The return value is a reference to the resource being reserved; you will use this reference to cancel the reservation.

Reserve a resource in a specified module.

TCA_REF_PTR
tcaReserveModResource(*modName*, *resName*)
char **modName*, **resName*;

Resources on different modules are allowed to have the same name, so tcaReserveModResource() is used in cases where *resName* is an ambiguous resource name. The module name *modName* specifies the appropriate module. In all other respects, this procedure is identical to tcaReserveResource().

Cancel a Resource Reservation

void
tcaCancelReservation(*ref*)
TCA_REF_PTR *ref*;

Cancels the reservation request represented by *ref*, where *ref* is the value returned by either tcaReserveResource() or tcaReserveModResource(). If *ref* does not represent a valid reservation reference, the calling module will be terminated, and the central server will log an error.

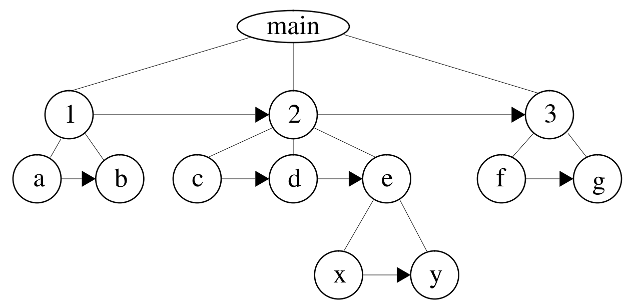
Task management

Controlling task execution order

Many computer programs are built on the assumption that procedures are executed sequentially. If a program makes a procedure call, it cannot proceed to another procedure call until the first call has returned. The following program shell is typical:

```
main()
{
    task-1();
    task-2();
    task-3();
}
task1()
{
    subtask-a();
    subtask-b();
}
task2()
{
    subtask-c();
    subtask-d();
    subtask-e();
}
subtask-e()
{
    sub-subtask-x();
    sub-subtask-y();
}
task3()
{
    subtask-f();
    subtask-g();
}
```

The figure on the right shows an execution graph of this program. We will refer to such a graph as a "task tree." In the graph, child nodes represent subtasks of the parent node. Nodes to the left of horizontal arrows must complete before nodes to right. For example, the task tree indicates that task 1 will finish completely before task 2 starts. Similarly, sub-subtask **x** cannot begin before subtask-d has finished.

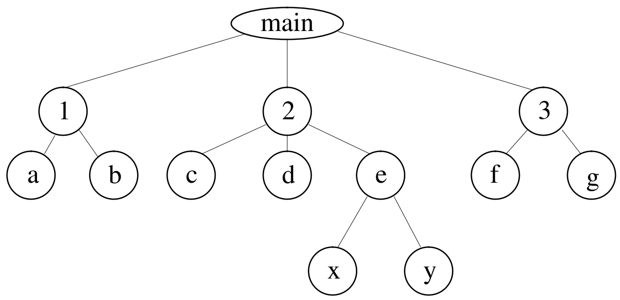


In many applications, particularly in robot control, this model is inadequate. Tasks 1, 2, and 3 might be completely independent: It may be possible to execute them on three different processors. Why make tasks 2 and 3 wait for task 1?

Suppose that we completely separate the subroutine call from the task execution. Assume that we can trigger the achievement of a task by sending a message. A routine that handles the message would be responsible for carrying out its action, or for sending additional messages.

```
main()
{
    send message 1
    send message 2
    send message 3
}
msg_1_hnd()
{
    send message a
    send message b
}
(etc.)
```

We send message 1 to trigger the achievement of task 1. We then immediately send message 2 to trigger the achievement of task 2. Note, however, that we do not wait for task 1 to complete. We simply send task 1's trigger message and proceed. The task tree changes somewhat; we remove all of the horizontal arrows showing temporal precedence. According to our new program, the order of task completion is not restricted in any way.



Each branch is independent of any other; any path down the tree may be followed without any concern for other paths. Node **x** might finish before node **b**; nodes **f** and **a** could begin at the same time.

If each path from root to leaf is completely independent of any other and no tasks make simultaneous demands on some resource, we have a very efficient, concurrent program. Otherwise, we have potential chaos, because we appear to have no control over the execution order or over access to system resources.

Ideally, we should be able to impose any needed *temporal constraints* (restrictions on the order of execution) on pairs of nodes—and remove constraints when they are not necessary. It should be possible, for example, to specify that task **c** must finish before task **y** can begin, but unnecessary orderings shouldn't be required.

Planning vs. execution

In robot control, as well as in other applications, it is useful to distinguish between plans and actions, because it is often practical to be able to *plan* a future task while executing the current task. A specific action might require a great deal of prior computation or information gathering before it can take place. It might not be possible to execute the action until a prior action is finished, but we may save a great deal of time if the *planning* for the action is finished (or at least under way) when it's time to execute the action. That is, if it is *permissible* to plan the next action while the current action is executing, the software architecture should make it *possible*.

Suppose, for example, that the task tree below represents a robot control program. The circular nodes correspond to *goal* nodes, or nodes in which only planning activity takes place. The square nodes—all leaves—correspond to robot commands (movement, picture-taking, etc.)

This graph specifies that commands **p** and **q** must be executed sequentially. That is, the command **p** must finish before the command **q** can being executing. Note, however, that the graph does not say anything about the nodes **e** and **x** with respect to node **p**. The planning activity related to nodes **e** and **x** can proceed before the command **p** finishes—perhaps before it even begins.

TCA allows you to make relationships between planning and execution explicit. For example, you can trivially specify these constraints:

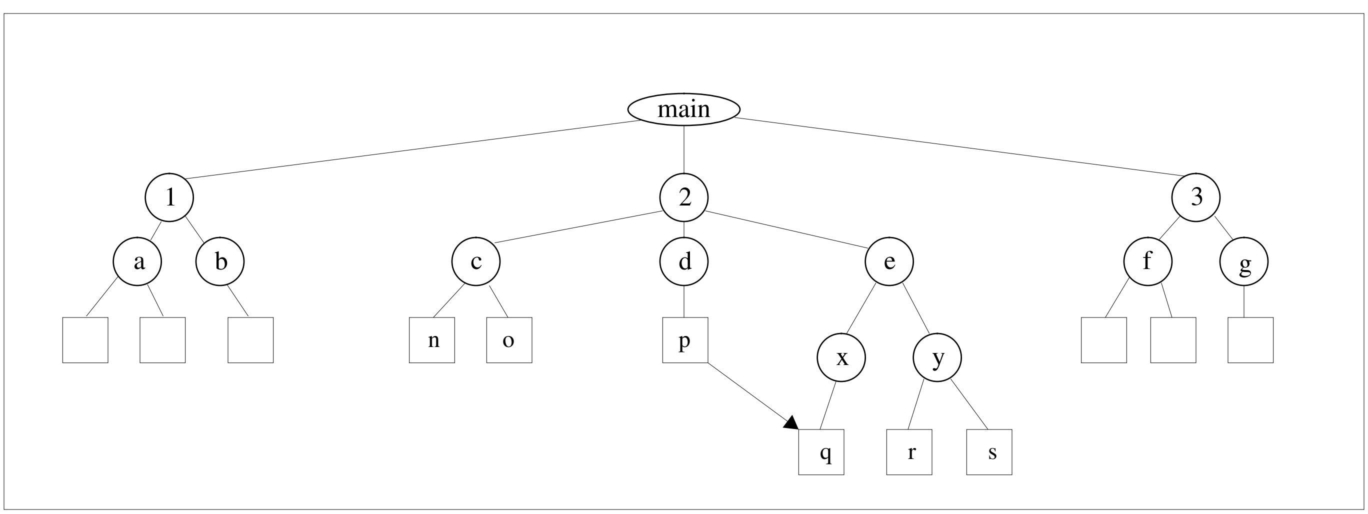
- all commands under a given node must finish before any commands under another given node can begin;
- all planning under a given node must finish before any planning under another given node can begin;
- all planning under a node has to be finished before any commands under the node can execute;

Goal messages and the creation of task trees

TCA's central server creates task trees when it forwards certain messages, and you can manipulate these trees. You may apply temporal constraints as needed, and you may destroy or re-create parts of the tree if necessary.

The building of task trees centers around *Goal* messages, which represent intentions to perform tasks. The underlying model is one of planning via hierarchical task decomposition. The procedures that handle goal messages typically:

- perform computations related to the task;



- send other goal messages to trigger sub-goals;
- send command messages to trigger execution of actions.

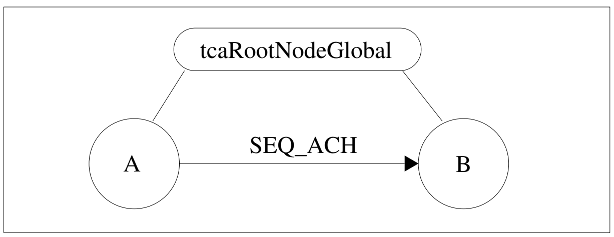
For an example, let us make the following assumptions:

- we have defined goal messages called “A,” “B,” “Q,” and “R,” along with command messages called “v,” “w,” and “x”;
- we have built and registered handlers for these messages, and they have names like “A_handler” and “x_handler”;

Suppose that from some top-level (a routine that is not a handler), we send the goal messages “A” and “B”:

```
main()
{
    …
    tcaExpandGoal(“A”, &A_data);
    tcaExpandGoal(“B”, &B_data);
    …
}
```

When the central server receives goal messages from a top-level procedure, the server creates nodes for these messages under a root node known as **tcaRootNodeGlobal**.

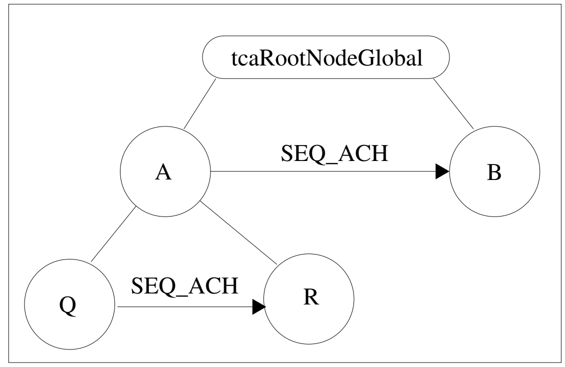


By default, the temporal constraint of *Sequential Achievement* (SEQ_ACH) is imposed on nodes A and B. This means that until all commands under A have been executed, no commands in B can be executed. Put another way, commands in A must precede commands in B, but there is no temporal restriction on the expansion (planning) of B.

Suppose that messages “Q” and “R” correspond to subgoals of “A,” so we send them from the handler associated with message “A”:

```
A_handler(ref, data)
{
    …
    tcaExpandGoal(“Q”, &Q_data);
    tcaExpandGoal(“R”, &R_data);
    …
}
```

Because the central server receives these goal messages from the handler of message “A,” corresponding goal nodes will be created under the node for “A”:



Once again, because we have not specified otherwise, the Sequential Achievement constraint is imposed on the new pair of nodes. No commands under R can begin until all commands under Q have finished.

Similarly, if B’s handler sends messages, nodes for these will be created directly under under the node for B. Note that query, multi-query, inform and broadcast messages are *not* added to the task tree.

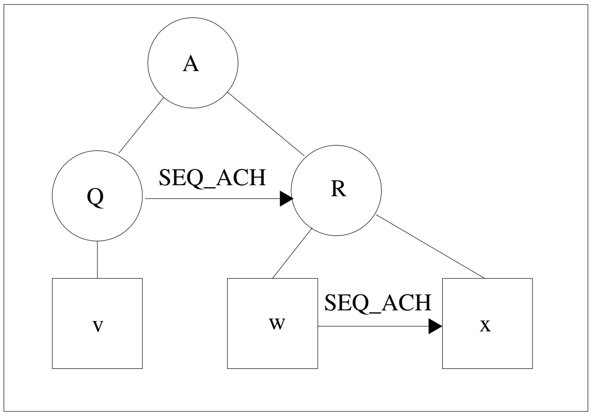
Finally, let us suppose that the handlers for messages Q and R send command messages:

```
Q_handler(ref, data)
{
    …
    tcaExecuteCommand(“v”, (void *)&v_data);
    …
}
.
.
R_handler(ref, data)
{
    …
    tcaExecuteCommand(“w”, (void *)&w_data);
    tcaExecuteCommand(“x”, (void *)&x_data);
    …
}
```

Any command messages sent from Q’s handler will cause command nodes to be created under the node for Q. Similarly, any command messages sent from R’s handler will cause command nodes to be created under the node for R.

While in this simple example, handlers issued either goal or command messages, in general any combination of goal and command messages can be sent within a single handler. A handler can even send a message to itself, enabling users to create recursive task trees.

The following diagram shows the task tree rooted at A:



Command nodes are represented differently from goal nodes because they are treated differently from goal nodes. This difference enables TCA to distinguish between planning a task and executing specific actions.

When the messages Q, R, v, w, and x have all been sent, we say that goal A has been completely *expanded* or *planned*. When all the handlers related to these messages have finished, we say that goal A has been *achieved*.

Specifications and descriptions for registering and sending goal messages follow:

Register a goal-class message.

TCA_RETURN_VALUE_TYPE
tcaRegisterGoalMessage(*msgName*, *msgForm*)
char **msgName*, **msgForm*;

Registers the string *msgName* as a valid goal message name. The string, *msgForm* is a data format string. See "Registering message data formats" on page 8 for more on format strings. The *msgForm* string may be NULL (NIL in LISP) to indicate that no data is to be sent with the goal.

Expand a goal.

TCA_RETURN_VALUE_TYPE
tcaExpandGoal(*goalName*, *data*)
char **goalName*;
void **data*;

Sends a goal message, using *data* as the data provided to the goal message handler's second parameter. The type of *data* depends on the particular message, but must correspond to the format string used in the associated tcaRegisterGoalMessage() call. The pointer may be NULL (NIL in LISP), indicating that no data is to be sent.

Returns a status value; currently Success is the only defined value.

Because goal messages are non-blocking, goal handlers don't reply to the issuing module. Instead, they issue "Success" or "Failure" messages (via tcaSuccess() or tcaFailure()) that are routed back to the central server. You can check on the current status of a goal message by using tcaReferenceStatus().

TCA issues a warning if a goal handler returns without having called tcaSuccess() or tcaFailure(); typically this warning should be heeded, because deadlock can occur if the central server is not informed that a goal message has finished processing. You can change this behavior by using tcaEnableDistributedResponses() (see page 14).

> The tcaExpandGoal() routine does not wait for a reply and returns immediately. Expanding a goal should not be confused with achieving a goal. The purpose of tcaExpandGoal() is to create a plan of action, not to execute the plan.

Creating placeholder nodes

If you want to apply temporal constraints to particular messages, it is often necessary to apply the constraints before sending the messages. Once messages have been sent, it is impossible to guarantee that constraints you place on them will be applied before their handling begins.

If you want the default temporal constraints applied to a message, you don't need to take any action; you can simply allow TCA to create a task tree node when the message is sent. Alternately, you can create "empty" nodes that aren't associated with particular messages; then, you apply temporal constraints to the nodes. Finally, you associate the nodes with specific messages and send the messages.

Suppose, for example that the handler for goal message A is being executed. This handler sends two other goal messages, G and H. It also specifies that the planning associated with message G must finish before the planning of message H can start. The following code implements this constraint.

```
A_handler(ref, data)
{
    TCA_REF_PTR Gref, Href;
    ...
    Gref = tcaCreateReference("G");
    Href = tcaCreateReference("H");
    ...
    tcaTplConstrain( tcaEndOf( tcaPlanningOf(Gref) ),
            "<", tcaStartOf( tcaPlanningOf(Href) ) );

    tcaExpandGoalWithConstraints(Gref, "G", G-data, NULL);
    tcaExpandGoalWithConstraints(Href, "H", G-data, NULL);
}
```

Create and return a reference to a message.

TCA_REF_PTR
tcaCreateReference(*msgName*)
char **msgName*;

Returns a reference to some future message *msgName* and adds a node to the task tree. The parent of the newly added node corresponds to the message that is currently being handled.

Note that tcaCreateReference() is just a special case of tcaAddChildReference().

If tcaCreateReference() is not called from within a handler, the new task tree node is added to tcaRootNodeGlobal, the top of the task tree. The created reference can be used to add temporal constraints using the routines tcaTplConstrain() or TPL_ORDER().

The new reference can be used to add temporal constraints using the routines tcaTplConstrain() or TPL_ORDER(), or in any other routine that takes a TCA_REF_PTR as an argument (such as task tree traversal routines).

Create and return a reference to a message, adding a new task tree node.

TCA_REF_PTR
tcaAddChildReference(*parentRefPtr*, *msgName*)
TCA_REF_PTR *parentRefPtr*;
char **msgName*;

Adds a task tree node as a child under the *parentRefPtr* node. (The reference *parentRefPtr* can be any valid reference to a task tree node.) With respect to the examples in "Temporal Constraint Constants" on page 28, the added node does not become the *previousNode* of the *parentRefPtr* until the message is actually sent (using tcaExpandGoalWithConstraints(), etc.).

In the following:

```
void fooHandler(ref, data)
TCA_REF_PTR ref;
char *data;
{
   TCA_REF_PTR newRef;
   ...
    newRef = tcaCreateReference(“bar”);
    tcaExpandGoalWithConstraints(newRef, “bar”, ...);
    ...
}
```

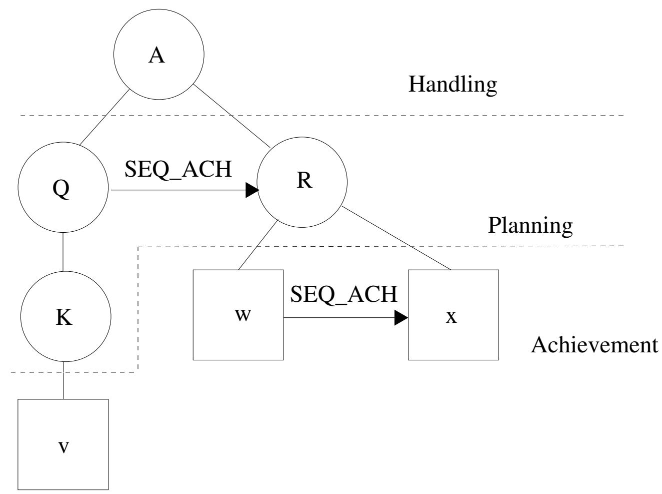
the call to tcaCreateReference() is equivalent to the call:

```
newRef = tcaAddChildReference(ref, “bar”);
```

The new reference can be used to add temporal constraints using the routines tcaTplConstrain() or TPL_ORDER(), or in any other routine that takes a TCA_REF_PTR as an argument (such as task tree traversal routines).

Temporal intervals of planning and execution

Any task tree rooted at a given node can be partitioned vertically into three sections with respect to that node: “handling,” “planning,” and “achievement.” These three extents are used by TCA to coordinate the planning and execution of tasks.



Three temporal intervals are associated with each node in the task tree:

- handling interval:
  Starts when message is forwarded to its handler. Ends when the handler finishes by sending a reply; the success or failure status of any message sent by the handler is not considered;
- planning interval (for goal messages only):
  Starts when message is forwarded to its handler. Ends when the handler finishes and all goal messages sent from the handler (and their subgoals, etc.) have been expanded down to commands;
- achievement interval:
  Starts when the first command message in the subree starts being handled. Ends when all messages under the node (including goal, command, monitor and exception messages) have been handled.

The following procedures return references to these intervals:

Get a reference to the handling interval of a message.

TCA_INTERVAL_TYPE
tcaHandlingOf(*msgRef*)
TCA_REF_PTR *msgRef*;

Returns a structure that refers to the handling interval of *msgRef*. This interval starts when the message is sent to a module, and ends when the associated handler responds (with Success/Failure).

Get a reference to the achievement interval of a message.

TCA_INTERVAL_TYPE
tcaAchievementOf(*msgRef*)
TCA_REF_PTR *msgRef*;

Returns a structure that refers to the achievement interval of *msgRef*. This interval is the time it takes to completely handle all the messages (goals, commands, monitors, and exceptions) of a subtree. The achievement interval of a leaf node in the task tree equals its handling interval.

The achievement interval of a goal is the union of the achievement intervals of all its subgoals, commands, and monitors.

Get a reference to the planning interval of a message.

TCA_INTERVAL_TYPE
tcaPlanningOf(*msgRef*)
TCA_REF_PTR *msgRef*;

Returns a structure that refers to the planning interval of *msgRef*. The planning interval of a goal is the time it takes to expand the task tree rooted at that goal, all the way down to, but not including, its executable primitives (commands and monitors). The start of the planning interval of a goal equals the start of its handling interval (i.e., a goal starts being planned when it is handled).

Only goals have meaningful planning intervals.

The preceding three routines are intended to be used in combination with the next two routines:

Get a reference to the starting time-point of an interval.

TCA_TIME_POINT_TYPE
tcaStartOf(*Interval*)
TCA_INTERVAL_TYPE *Interval*;

Returns a structure that refers to the time at which *Interval* begins, where *Interval* is a handling, achievement, or planning interval.

Get a reference to the ending time-point of an interval.

TCA_TIME_POINT_TYPE
tcaEndOf(*Interval*)
TCA_INTERVAL_TYPE *Interval*;

Returns a structure that refers to the time at which *Interval* ends, where *Interval* is a handling, achievement, or planning interval.

Temporal Constraint Constants

TCA provides several pre-defined constants for temporal constraints. Routines such as tcaExpandGoalWithConstraints() (see below) take a "temporal constraint" argument, and these constants are suitable values.

Combinations of the defined constraints are permitted. For example, you would specify sequential achievement with delayed planning by using the constraint SEQ_ACH+DELAY_PLANNING.

The descriptions below assume that inside some handler you want to call a procedure that will send a message with constraints (e.g. tcaExpandGoalWithConstraints()), and the last goal, command, or point monitor message sent from that handler has a node in the task tree, *previousNode*. The node that will be created by the procedure you're calling is *thisNode*.

- SEQ_ACH
  Specifies the temporal constraint:

  tcaEndOf(tcaAchievementOf(*previousNode*)) <=
   tcaStartOf(tcaAchievementOf(*thisNode*))

  The achievement of *previousNode* must be completed before starting achievement of *thisNode*. That is, all commands under *previousNode* must finish before any commands under *thisNode* start. If you place this constraint on the first message sent by a handler, the constraint has no effect.

  Usually, there is no point in using SEQ_ACH by itself, because it is the default temporal constraint used by the system. All messages carry the SEQ_ACH constraint by default.

- SEQ_PLANNING
  Specifies the temporal constraint:

  tcaEndOf(tcaPlanningOf(*previousNode*)) <=
   tcaStartOf(Planning(*thisNode*))

  The planning of *previousNode* must be completed before the start of the planning of *thisNode*. That is, *previousNode* must be completely expanded before *thisNode* starts being handled. If this is the first message sent by a handler, the constraint has no effect. It is only meaningful when both *previousNode* and *thisNode* are goal messages.

- PLAN_FIRST
  Specifies the temporal constraint:

  tcaEndOf(tcaPlanningOf(*thisNode*)) <=
   tcaStartOf(tcaAchievementOf(this*Node*))

  The task tree rooted at *thisNode* must be completely expanded before any command or monitor in the tree can start executing. Only applicable to goal messages.

- DELAY_PLANNING
  Specifies the temporal constraint:

  tcaStartOf(tcaPlanningOf(*thisNode*)) =

tcaStartOf(tcaAchievementOf(*thisNode*))

The message associated with *thisNode* will not be expanded until execution time (the time when it is ready to be achieved). This constraint is only meaningful for goal messages.

Note that it is inconsistent to specify both PLAN_FIRST and DELAY_PLANNING for the same goal.

Specifiying SEQ_ACH+DELAY_PLANNING for every goal message gives you a completely serial system, corresponding to the task tree on page 23.

These pre-defined constants are the most commonly needed temporal constraints. The following two procedure specifications show examples of how you may use the constraint constants.

Expand the current node using the specified temporal constraints.

TCA_RETURN_VALUE_TYPE
tcaExpandGoalWithConstraints(*ref*, *name*, *data*, *tplConstr*)
TCA_REF_PTR *ref*;
char **name*;
void **data*;
int *tplConstr*;

ref is a message reference, created by a call to tcaCreateReference() or tcaAddChildReference(). *ref* must be a reference previously created for the message *name*. If *ref* is NULL, the system creates a new reference. *name* is the name of the goal message, and *data* is the data to be sent with the message. *tplConstr* is an expression combining the pre-defined temporal constraints described above (e.g "SEQ_ACH+DELAY_PLANNING"). If no constraints are needed, *tplConstr* should be 0.

Note:

```
tcaExpandGoal(name, data);
```

is equivalent to

```
tcaExpandGoalWithConstraints(NULL, name,
        data, SEQ_ACH);
```

Execute a command following the specified temporal constraints (non-blocking)

TCA_RETURN_VALUE_TYPE
tcaExecuteCommandWithConstraints(*ref*, *name*, *data*, *tplConstr*)
TCA_REF_PTR *ref*;
char **name*';
void **data*;
int *tplConstr*;

ref is a message reference, created by a call to tcaCreateReference() or tcaAddChildReference() *ref* must be a reference previously created for the message *name*. If *ref* is NULL, the system creates a new reference. *name* is the name of the command message, and *data* is the data to be sent with the message. *tplConstr* is an expression combining the pre-defined temporal constraints described above. If no constraints are needed, *tplConstr* should be 0.

Note:

```
tcaExecuteCommand(name, data);
```

is equivalent to

```
tcaExecuteCommandWithConstraints(NULL, name,
        data, SEQ_ACH);
```

Execute a command following the specified temporal constraints (blocking)

TCA_RETURN_VALUE_TYPE
tcaWaitForCommandWithConstraints(*ref*, *name*, *data*, *tplConstr*)
TCA_REF_PTR *ref*;
char **name*;
void **data*;
int *tplConstr*;

This is a blocking form of tcaExecuteCommandWithConstraints(). The command, *name*, is issued and the call waits until the command replies by calling tcaSuccess() or tcaFailure(). If the handler finishes with a call to tcaSuccess(), then the return value is Success; otherwise the return value is Failure.

Note:

```
tcaWaitForCommand(name, data);
```

is equivalent to

```
tcaWaitForCommandWithConstraints(NULL, name,
        data, SEQ_ACH);
```

Specifying Constraints

A module can impose temporal constraints on the intervals associated with task tree nodes to order the planning and execution of different nodes. You can use the defined constants described in the preceding section, or, if you need greater flexibility, you can use more explicit temporal mechanisms provided by TCA.

Suppose that you must specify that command A must *start* before command B, but there is no other restriction, that is, command A doesn't have to *finish* before command B. No combination of the constraint constants can express this condition. The following functions provide capabilities beyond the constraint constants.

Assert temporal relationship between two time-points.

void
tcaTplConstrain(*time1*, *relationship*, *time2*)
char **relationship*;
TCA_TIME_POINT_TYPE *time1*, *time2*;

Specifies a temporal constraint between *time1* and *time2*.

Values for *time1* and *time2* are obtained using the calls tcaStartOf() and tcaEndOf(). The value for *relationship* is a string, and can be one of "<=", "<", ">=", ">", "=". tcaTplConstrain() is used to add constraints between task tree nodes.

Note that while TCA will always maintain inequality constraints (e.g., goal G2 ends before G3 begins) the equality constraint "=" (e.g. command C3 and C4 start at the same time), may not be strictly adhered to, depending on resource availability and the vagaries of the communications network. Thus, while inequality constraints provide reliable synchronization, equality constraints should not be relied upon for operations where tight synchronization is needed.

Suppose, for example, that the handler for a goal message called "fixit" is being executed. This handler sends two other goal messages—one to get the appropriate tools, and one to fix some object. However, the system must plan how to fix the object before it can decide which tools are appropriate. Thus, the achievement of "get tools" must preced the achievement of "fix object," but the planning occurs in the opposite order. The following code implements this constraint:

```
fixit_handler(ref,data)
TCA_REF_PTR ref;
DATA_TYPE data;
{
    TCA_REF_PTR fixRef, getRef;

    fixRef = tcaCreateReference("fix object");
    getRef = tcaCreateRererence("get tools");

    tcaTplConstrain( tcaendOf( tcaPlanningOf(fixRef)),
            "<=",
            tcaStartOf( tcaPlanningOf(getRef)) );

    tcaExpandGoalWithConstraints(getRef, "get tools",
            &toolData, 0);

    tcaExpandGoalWithConstraints(fixRef, "fix object",
            &fixData, SEQ_ACH);
}
```

Order one interval to precede another.

(macro—no return value)
TPL_ORDER(*interval1*, *interval2*)
TCA_INTERVAL_TYPE *interval1*, *interval2*;

Specifies that *interval1* must precede *interval2*. This macro is defined as:

```
tcaTplConstrain( tcaEndOf(interval1), "<=",
        tcaStartOf(interval2) )
```

tcaDelayCommand

TCA provides a server-handled command message that waits an integer number of seconds before returning success. TCA pre-registers this message, using this definition:

```
tcaRegisterCommandMessage("tcaDelayCommand", "int");
```

For example:

```
delay = 20;
tcaExecuteCommand("tcaDelayCommand", &delay);
```

This will create a command node that will wait 20 seconds before returning Success. Typically, this message is used to insert pauses into plans of action.

Operations on Task Trees

The procedures presented on the next few pages allow you to traverse, display, and prune task trees. They are intended to be used to handle exceptions, to implement replanning operations, and possibly to implement interfaces that give the user the capability of adding to or changing the plans made the system.

Traversing Task Trees

These routines enable a module to traverse the task tree. Given a reference to a node in the task tree (a variable of type TCA_REF_PTR), the task tree traversal routines enable you to find the parent, children, and siblings of the given node (or more generally, the ancestors and descendants of the node).

Note that these routines return a reference that can be used in other traversal, display, manipulation, and temporal constraint routines. You can retrieve the message name and data associated with these references by calling tcaReferenceName() and tcaReferenceData().

Get a reference to the parent of a task tree node.

TCA_REF_PTR
tcaFindParentReference(*childRefPtr*)
TCA_REF_PTR *childRefPtr*;

Returns a reference to the parent node of *childRefPtr*, which is some node in the task tree. The procedure will return NULL if *childRefPtr* has no parent. This can occur either if it is the top node of the tree (tcaRootNodeGlobal), or if the child reference was issued by a query or constraint message. (Recall that queries and constraint messages do not generate nodes in task trees.)

Get a reference to the “first” child of a node.

TCA_REF_PTR
tcaFindFirstChild(r*efPtr*)
TCA_REF_PTR *refPtr*;

Returns a reference to the child node of *refPtr* that must be achieved before any other child of *refPtr*. Typically, this will be the node associated with the first message sent from the handler associated with *refPtr*, but temporal constraints determine which child is actually “first.” Sometimes, the “first” child will be ambiguous (when sibling nodes can be achieved concurrently). In such cases, one of the nodes is arbitrarily chosen as the “first” child.

Get a reference to the “last” child of a node.

TCA_REF_PTR
tcaFindLastChild(r*efPtr*)
TCA_REF_PTR *refPtr*;

Returns a reference to the child node of *refPtr* that must be achieved after all other children of *refPtr* have been achieved. Typically, this will be the node associated with the last message sent from the handler associated with *refPtr*, but temporal constraints determine which child is actually “last.” Sometimes, the “last” child will be ambiguous (when sibling nodes can be achieved concurrently). In such cases, one of the nodes is arbitrarily chosen as the “last” child.

Get a reference to the “previous” sibling node.

TCA_REF_PTR
tcaFindPreviousChild(*childRefPtr*)
TCA_REF_PTR *childRefPtr*;

Returns a reference to the sibling node that must be achieved immediately before *childRefPtr*. Returns NULL if *childRefPtr* is the first sibling node.

Get a reference to the “next” sibling.

TCA_REF_PTR
tcaFindNextChild(*childRefPtr*)
TCA_REF_PTR *childRefPtr*;

Returns a reference to the sibling node that must be achieved immediately after *childRefPtr*. Returns NULL if *childRefPtr* is the last sibling node.

Get a reference to the child node associated with a certain message.

TCA_REF_PTR
tcaFindChildByName(r*efPtr*, *msgName*)
TCA_REF_PTR *refPtr*;
char **msgName*;

Returns a reference to the first child node of *refPtr* that is associated with the message *msgName*. Returns NULL if no such child exists.

Get a reference to the highest parent node in the task tree for a given node.

TCA_REF_PTR
tcaFindTopLevelReference(*childRefPtr*)
TCA_REF_PTR *childRefPtr*;

Returns the ancestor node of *childRefPtr* whose parent is tcaRootNodeGlobal. The procedure returns NULL if *childRefPtr* is itself tcaRootNodeGlobal, or if *childRefPtr* is not a valid task tree node.

You might use this routine, for instance, if you want to terminate the top-level task when an unrecoverable error is detected at some lower level of the task tree.

Get a reference to an ancestor node by message name.

TCA_REF_PTR
tcaFindAnteReferenceByName (*childRefPtr*, *anteName*)
TCA_REF_PTR *childRefPtr*;
char **anteName*;

Returns the first ancestor of *childRefPtr* whose associated message is *anteName*. Returns NULL if no ancestor node has that message name.

Get a reference to a node whose message generated an exception-class message node.

TCA_REF_PTR
tcaFindFailedReference(*ERefPtr*)
TCA_REF_PTR *ERefPtr*;

Searches up the task tree to return the first node that does not refer to an exception message. *ERefPtr* must be a reference to an exception message (the reference passed as the first argument to an exception handler).

This function is used primarily for exception handling, to find the node that issued the failure. Typically, a decision on how to handle the failure will be made by examining the node that issued the failure message. Options for handling the failure include killing or retrying the failed node, or adding new references to the tree to perform “replanning”. See the chapter "Exception handlers" on page 39.

Test whether two references identify the same task tree node.

int
tcaIsSameReference (*refPtr1*, *refPtr2*)
TCA_REF_PTR *refPtr1*, *refPtr2*;

Returns TRUE (1) if the node references *refPtr1* and *refPtr2* refer to the same task tree node, FALSE (0) otherwise. This procedure can be used, for instance, to check if two nodes have the same parent

```
tcaIsSameReference (
   tcaFindParentReference(childRefPtr1),
   tcaFindParentReference(childRefPtr2)  )
```

Note: The task tree traversal routines will not return the exact same TCA_REF_PTR each time; thus, one should always use tcaIsSameReference() to test for equality.

Get the name of the message associated with a particular reference.

char *
tcaReferenceName(*ref*)
TCA_REF_PTR *ref*;

If a single handler services different messages, the handler must be able to tell which of those messages triggered its invocation. This routine enables a handler to determine the message name so that it can properly interpret the message data.

Get the data of the message associated with a particular reference.

char *
tcaReferenceData(*ref*)
TCA_REF_PTR *ref*;

Enables one module to access the data associated with a message sent by another module. The reference is typically obtained via task tree traversal routines.

This routine is particularly useful for replanning operations, that is, when you need to alter the data associated with a message before retrying it.

Displaying Task Trees

The developers of TCA hope someday to add tools that graphically display task trees. Currently, the only facility for viewing task trees prints out a listing of tree node names at the central server, with indentation indicating the hierarchy of the nodes.

Display all or part of a task tree.

void
tcaDisplayTaskTree(*dispRefPtr*)
TCA_REF_PTR *dispRefPtr*;

Displays a list of nodes in the task tree rooted at node *dispRefPtr*. Indentation is used to specify the hierarchy of the nodes. The listing shows the name of the message and indicates the message status. “A” means the message is active (being handled); “I” means temporally inactive; “P” is pending resource availability, and “C” is completed (has been handled).

This procedure uses the logging facility of the central server, so the output will go to the server’s terminal, the log file, or both, depending on what arguments you specify when you start the central server.

Pruning Task Trees

Destroy a task tree starting at a particular node.

void
tcaKillTaskTree(*killRefPtr*)
TCA_REF_PTR *refPtr*;

Terminates the node *killRefPtr* and all its descendant nodes (goals, commands, and monitors).

Note that TCA cannot interrupt a module; if a message is being handled by a module, TCA will let it continue to completion before killing the associated node. This means that some message handling may continue after a task tree is killed. However, any goal, command or monitor message issued by these "in progress" handlers will be immediately killed and not be handled at all. On the other hand, queries issued by these "in progress" handlers will be handled in order to allow the handlers to complete normally.

Destroy the descendants of a task tree starting at a particular node.

void
tcaKillSubTaskTree(*killRefPtr*)
TCA_REF_PTR *killRefPtr*;

Terminates the children of *killRefPtr* and all their descendant nodes. As with tcaKillTaskTree(), messages that are currently being handled will be allowed to complete.

This function could be used in error recovery for removing a planned sequence of actions and replanning (e.g., by using tcaRetry() or by using tcaAddChildReference() to add a new node under the *killRefPtr*, which will then be expanded when the actual message is sent).

Clean up the Task Tree after message is achieved.

TCA_RETURN_VALUE_TYPE
tcaCleanUpAfterAchieved(*msgName*)
char **msgName*;

Kills the task subtree associated with *msgName* after each instance of the message has been achieved. The message must be a goal, command, or monitor class message.

Another way of saying this is that when all nodes in the message's subtree have been handled, the subtree is deleted.

Release a reference to a particular node.

void
tcaReleaseReference(*refPtr*)
TCA_REF_PTR *refPtr*;

Indicates that the reference *refPtr*, which was obtained from a previous TCA call (e.g. tcaFindNextChild()) is no longer in use.

After a reference has been released, it is no longer valid and cannot be used in subsequent TCA routines.

This routine is necessary because of the distributed nature of TCA: One module may be traversing a task tree while another module is trying to terminate it. To prevent conflict, the node associated with a reference does not actually "go away" (no memory is freed) until all modules that have received a copy of that reference (e.g. by using tcaAddChildReference() or the tree traversal routines) have released the reference.

Monitors

A monitor tests for a condition and takes some action if the condition holds. Monitor messages are somewhat unusual because you don't write handler routines for them. Rather, you specify the handling of a monitor message by providing TCA with two other message names: a "condition message," which must be a query message, and an "action message," which may be a goal, command, or monitor message.

When TCA's central server receives a monitor message, it delivers that monitor's condition message, subject to any temporal constraints. If the query returns the value NullReply, no action is taken. Otherwise, TCA delivers the action message. TCA uses the query's reply data as the data for the action message.

Suppose that you have defined messages and handlers for the following:

- a query to check the voltage level of your robot's battery; if the voltage is above a certain threshhold, the query retuns NullReply; otherwise, it returns the voltage as its reply data.
- a goal that terminates the current task and plans a battery recharging operation;

You could define a battery voltage monitor using these two messages: the first as the query message, and the second as the action message. Whenever one of the monitor's queries returns a voltage level, TCA will send the action message, which will schedule a recharge.

TCA provides two types of monitors: point monitors and interval monitors. Point monitors test a condition only once, so the associated action message will be sent at most once. A point monitor can be thought of as an "if-then" test. The query message (the "if" condition) is sent, and unless the return value to the query is NullReply, an action message (the "then" action) is sent, along with the data sent in response to the query. You can implement more complex conditionals, such as *if-then-else* or *case* tests, by having the query always return data, and using the action message to discriminate among the various cases.

Point Monitors

Point monitors are convenient for testing pre-conditions or post-conditions for tasks. For example after executing a "follow path" task, a point monitor may query to see if the robot has successfully arrived at the desired location and take the action of replanning the path if the original plan failed.

Point monitors are defined using tcaRegisterPointMonitor(). Point monitor messages are sent by the tcaPointMonitor() or tcaPointMonitorWithConstraints() routines. Temporal constraints can be placed on the execution of point monitors using the mechanisms described earlier. (See "Controlling task execution order" on page 23.)

Register a Point Monitor.

void
tcaRegisterPointMonitor(*monName*, *condName*, *actName*)
char **monName*, **condName*, **actName*;

Creates a point monitor called *monName* that uses *condName* as its condition message and *actName* as its action message. When the monitor is triggered, the query message *condName* is sent with the data passed to the call that initiated the monitor (e.g. tcaPointMonitor()). Unless the *condName* message returns with the value NullReply, an *actName* message is sent with the data of *condName*'s reply.

The monitor can be registered before *condName* and *actName* have been registered. The condition message must be a query class message, while the action message must be a goal, command, or monitor class message. The reply data format of *condName* message must be the same as the data format of *actName*.

Initiate a Point Monitor.

TCA_RETURN_VALUE_TYPE
tcaPointMonitor(*monName*, *data*)
char **monName*
DATA_TYPE **data*;

Issues a point monitor *monName*. *data* is the data passed to the condition query of the monitor, and its type depends on the message type. It should be of the same format as that described when registering the monitor's condition message. Unless the return value of the condition query is NullReply, the reply data is forwarded to the action message.

Note that because tcaPointMonitor() is non-blocking, the monitor has not necessarily been executed when the routine returns control to the module issuing the monitor. A point monitor is not actually triggered until its temporal constraints are met. The call tcaPointMonitor() places the SEQ_ACH temporal constraint on the monitor, that is, it is activated when the previous goal, command, or monitor issued by the handler has been achieved. If the point monitor is the first such message issued by the handler, then the point monitor is activated when its parent is ready to be achieved.

Initiate a Point Monitor with temporal constraints.

TCA_RETURN_VALUE_TYPE
tcaPointMonitorWithConstraints(*mRef*, *monName*, *data*, *tc*)
TCA_REF_PTR *mRef*;
char **monName*
DATA_TYPE **data*;
int *tc*;

Similar to tcaPointMonitor()—issues a point monitor *monName*, with *data* being passed to the monitor's condition query. The difference is that tcaPointMonitorWithConstraints() allows you to overide the default SEQ_ACH constraint inherent in a tcaPointMonitor() call.

mRef is a reference to a message, created by a call to tcaCreateReference(). If *mRef* is NULL, the system creates a new reference for the monitor message. *tc* is a set of temporal constraints formed from the pre-defined constraints provided by TCA.

Interval Monitors

Interval monitors remain active over an interval that you specify; they detect changes over time. Like point monitors, interval monitors check for some condition and send an action message if the condition holds. Unlike point monitors, interval monitors can check a condition (and send an action message) many times during its active interval.

Examples of interval monitors used by a particular robot system include detecting low battery charge (which is active when the robot is off its charger), and detecting the appearance of new obstacles in a segment of the robot's path (which is active starting when the robot makes the plan and ending when the robot has traversed the segment).

To initiate an interval monitor, send a message with tcaIntervalMonitor(); this routine specifies the monitor name and data to be sent to the condition message. Alternatively, the call tcaIntervalMonitorWithConstraints() provides additional options to the monitor, including overriding the default temporal constraints (similar to the other "WithConstraints" routines), specifying start and end times relative to other tasks, and other options controlling the length and rate of sending condition and action messages.

TCA has two variations of interval monitors: polling and demon monitors.

Polling Monitors

Polling monitors are defined by the call tcaRegisterPollingMonitor(), which defines the monitor's name, a condition, and action message. After you send a polling monitor message, TCA sends the condition message, along with the data provided in the tcaIntervalMonitor() call that issued the monitor. Unless the condition's return value is NullReply, the action message is sent. Once the condition query has returned, the polling monitor will wait a specified period before sending the condition message again— the actual period may be slightly longer than the specified period, given the factors of resource contention and message traffic through the central server. Note that the monitor can optionally be set to terminate after a specified number of action messages have been sent; typically 1 or infinity are the preferred choices for this option.

Register a Polling Monitor.

void
tcaRegisterPollingMonitor(*monName*, *condName*, *actName*)
char **monName*, **condName*, **actName*;

Registers the point monitor *monName*, having the condition message *condName* and the action message *actName*.

Both the *condName* and *actName* messages need not be registered before the monitor is registered. *condName* must be a query class message, and *actName* must be a goal, command, or point-monitor class message. The reply data format of the *condName* message must be the same as the data format of the *actName* message.

Demon Monitors

Demon (interrupt-driven) monitors are defined by the call tcaRegisterDemonMonitor(), by which you define the demon's name, a "set-up message," an action message, and a cancellation message. Demon monitors, like polling monitors, are active for an interval period, but unlike polling monitors, they asynchronously alert the central server when the action message should be sent. When the start time for a demon monitor has arrived the central server sends the set-up message (a query) for the monitor. The set-up message handler should initialize the monitor within a module and return a unique integer ID with which to identify that monitor. The set-up message, which is sent the data provided in the call to tcaIntervalMonitor(), basically serves the same purpose as the condition message of the other types of monitors. When a module determines that a demon's action message should be sent it notifies the central server using the call tcaFireDemon(). When the monitor terminates the demon's cancel message is sent, with the data being the id returned by the set-up message. After the cancel message has been sent, subsequent calls to tcaFireDemon() will be ignored.

For example, a particular robot's vision module continually takes and processes images. A demon monitor is set up to check for obstructed paths. The set-up message handler accepts a segment of a path, generates a unique identifier for the monitor, places it in its list of monitors, and replies with the id number. Then, whenever a new image is processed, the vision module checks each segment in the list to see if it is obstructed, and if so issues a tcaFireDemon() call. Finally, when a cancellation message is received, the vision module deletes the element in its list with the given id.

Register a Demon Monitor.

void
tcaRegisterDemonMonitor(*monName*, *setUpName*, *actName*, *cancelName*)
char **monName*, **setUpName*, **actName*, **cancelName*;

Registers a demon monitor called *monName*, having the set-up message *setUpName*, the action message *actName*, and the cancel message *cancelName*.

The *setUpName*, *actName*, and *cancelName* messages need not be registered before the monitor is registered. *setUpName* must be a query class message, whose reply data format is “int” (the integer identifier of the monitor). *actName* must be a goal, command, or monitor class message. *cancelName* must be a constraint class message, and its data format must be “int” (the identifier of the monitor to be canceled).

Request that a demon monitor’s action message be sent.

TCA_RETURN_VALUE_TYPE
tcaFireDemon(*monName*, *demonUserId*, *data*)
char **monName*, **data*;
int *demonUserId*;

When a module determines that a monitor’s condition holds, *tcaFireDemon* is used to inform TCA to issue the monitor’s action message. *monName* is the name of the monitor. *demonUserId* is the integer identifier that the user’s code generated in response to the monitor’s set-up message. *data* is the data to be forwarded on to the action message—it should be of the same format as that described when registering the monitor’s action message. tcaFireDemon() is non-blocking, so the user should not assume that the action message was handled when the routine returns. tcaFireDemon() should not be called with a given *demonUserId* after a cancellation message is received for that identifier.

Create and initialize a monitor option structure

INTERVAL_MONITOR_OPTIONS_PTR
tcaCreateMonitorOptions()

```
typedef struct {
    int maxFire;
    int fireEvery;
    int duration;
    int period;
    int initialWait;
} INTERVAL_MONITOR_OPTIONS_TYPE,
    *INTERVAL_MONITOR_OPTIONS_PTR;
```

This routine creates a structure initialized with default values specified below. It returns a pointer to the structure. You can alter the values in this structure and then use the pointer as an argument to the routine tcaIntervalMonitorWithConstraints().

- maxFire:
  The maximum number of times to fire action messages. A value of -1 indicates infinite. The default is -1. For example, setting maxFire to 1 means the monitor terminates after the first time the action message is sent (i.e. the first time the condition message returns a non-null reply).

- fireEvery:
  Fire the action message every *n* times the condition is satisfied. The default is 1.

- duration:
  The maximum time, in seconds, that a monitor can run. The value of -1 indicates infinite time. The default is -1.

- period:
  The polling period in seconds. The default is 10 seconds.

- initialWait:
  TRUE or FALSE. If TRUE (1) then wait for *period* seconds before sending the first condition message. If FALSE (0) the condition query is sent as soon as the monitor becomes active. The default value is FALSE.

Initiate an Interval Monitor (Polling/Demon) with Possible Constraints.

TCA_RETURN_VALUE_TYPE
tcaIntervalMonitorWithConstraints(*ref*, *name*, *data*, *start*, *end*, *options*)
TCA_REF_PTR *ref*;
char **name*
DATA_TYPE **data*;
TCA_TIME_POINT_TYPE *start*, *end*;
INTERVAL_MONITOR_OPTIONS_PTR *options*;

This is the most general routine for issuing an interval monitor. It is used for both polling and demon monitors.

The monitor to issue is *name*. *data* is the data passed to the condition query of the monitor—it should be of the same format as that described when registering the monitor’s condition message.

ref is a reference for use with temporal constraints. If *ref* is NULL, then TCA will create a reference.

start is the time point when the monitor should be activated, (i.e. when the monitor starts polling or when the set-up message is sent). The start time is defined by its relationship to other tasks. If *start* is given the value **tcaDefaultTime()**, the monitor begins immediately. The parameter *end* is the time point when the monitor should terminate (i.e., polling is stopped or a cancellation message is sent). If *end* is given the value **tcaDefaultTime()**, the monitor ends when its parent node in the task tree is completely achieved. If the monitor was issued outside of a handler, then the default is to continue indefinitely.

The *options* parameter is of the form created by the routine tcaCreateMonitorOptions(). If a NULL value is passed to *options* then the default values are used.

For example, the following indicates that monitor "foo" should begin when message G1 is ready to be achieved, and continue until the task tree under message Ref has been completely achieved.

```
void baz_handler (Ref, data)
TCA_REF_PTR Ref;
char *data;
{   ...
    G1 = tcaCreateReference("G1");
    tcaIntervalMonitorWithConstraints(NULL, "foo", data,
            tcaStartOf(  tcaAchievementOf(G1)  ),
            tcaDefaultTime(), NULL);
     ...
}
```

Initiate an interval monitor with default options.

TCA_RETURN_VALUE_TYPE
tcaIntervalMonitor(*name*, *data*)
char **name*, **data*;

A simplified way to issue interval monitors, tcaIntervalMonitor() is equivalent to:

```
tcaIntervalMonitorWithConstraints(NULL, name, data,
     tcaDefaultTime(), tcaDefaultTime(), NULL);
```

Returns the default time to start/stop a monitor.

TCA_TIME_POINT_TYPE
tcaDefaultTime()

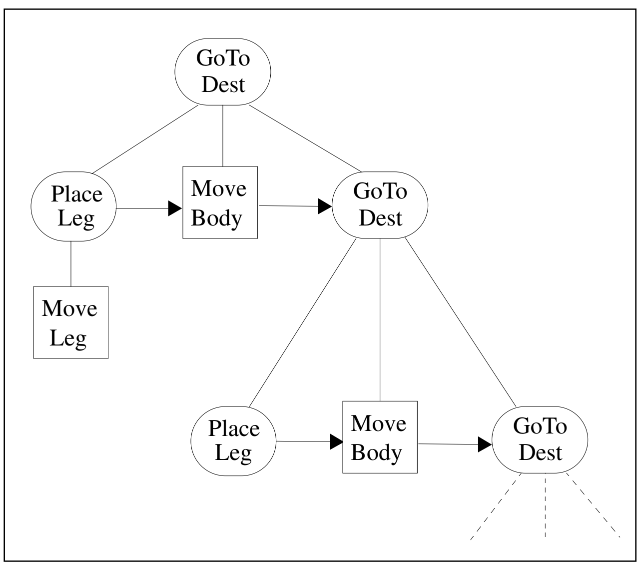
The default start and end times for an interval monitor. The default times are that the monitor will start when it is first issued (i.e., immediately), and will stop when its parent node is achieved. This function supplies those default values for use in **tcaIntervalMonitorWithConstraints()**.

Exception handlers

Suppose that a legged robot has a simple goal called "GoToDest" (for "Go To Destination.") To achieve this goal, the robot would make sure that its current position isn't the destination; then, it would carry out the recursive algorithm:

- Place Leg;
- Move Body;
- Go To Destination.

To illustrate, here is an appropriate task tree section. Arrows indicate a Sequential Achievment constraint.



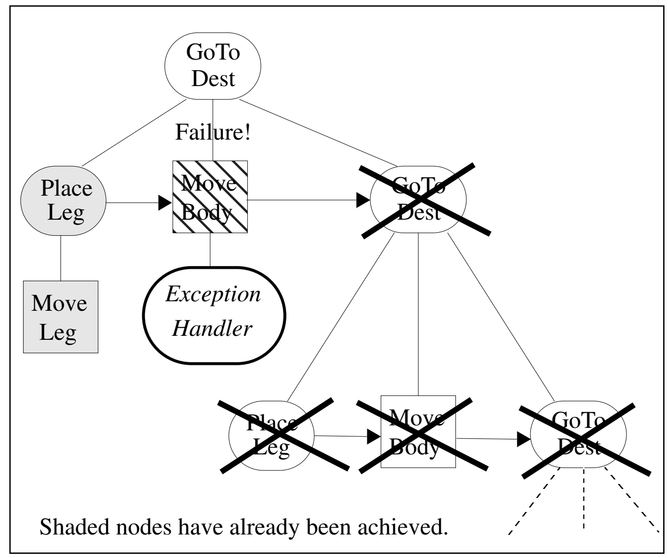
If all goes well, the robot will eventually reach the destination; no further "GoToDest" goals will be generated, and this part of the tree will eventually be achieved.

But what if all doesn't go well? Suppose that something were to go wrong with the first "Move Body" command? We would have two problems:

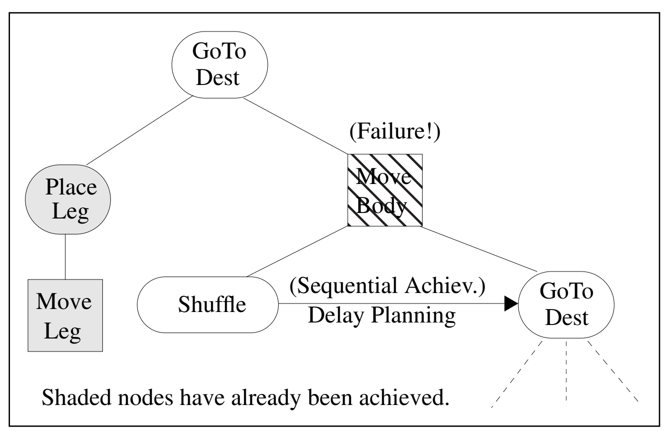
- We must try some recovery strategy;
- We must discard planned activity, because in this instance, the robot's state is not what we assumed it would be; our planning is therefore suspect.

What we need is a way to attach a failure recovery strategy to the "Move Body" node. If a failure were reported, the recovery procedure could take control. It must do the following:

First, find the next child after "MoveBody" and kill it (by using tcaKillTaskTree());



Second, execute an appropriate recovery procedure. One strategy is to "shuffle" the robot's legs into a standard configuration, and then to try to move again. Planning the next move must be delayed until the shuffling is finished. The "patched" tree would be this:



When we must make changes to the tree, we don't want to tamper with the basic walking algorithm, because we expect the "GoToDest" tree to give us good results in most cases. As much as possible, we want to separate the treatment of "normal" and "exceptional" behavior.

TCA's exception handling mechanism helps you to separate error recovery algorithms from the algorithms for "normal" behavior. You can associate error recovery procedures with particular task tree nodes. As you test your robot control software and discover exceptional situations, you can incrementally attach exception handling procedures to appropriate nodes in the task tree.

This incremental approach is particularly effective in robot control, because robot environments can be dynamic and thus unpredictable. It's also difficult, if not impossible, to predict all potential failure modes, and continual tinkering with the "normal" algorithm can make your code difficult to comprehend. TCA's exception handling mechanismas provide for incremental, modular adjustment.

Sending an exception message

To "raise an exception," that is, to send a message that will invoke an exception message handler, call tcaFailure() (see page 15). Calling tcaFailure() indicates that a planning-time (goal) or execution-time (command) failure has occurred, or that an exception has been detected by monitors.

Recall that the second parameter of tcaFailure() is a "description" of the failure. When this description is the name of a registered exception message, TCA sends that message, along with the "failure data" provided in the third parameter to tcaFailure().

For example:

```
.
.
typedef struct { int datum1, datum2} EXCEPTION_DATA;
EXCEPTION_DATA exception;

tcaRegisterExceptionMessage("We have a problem",
    "{int, int}");
.
.
tcaFailure("We have a problem", &exception);
.
.
.
```

TCA also creates a node in the task tree for the exception message and places it as the child of the node that raised the exception (see figure on the previous page).

Registering exception messages and handlers

Modules register exception messages by calling tcaRegisterExceptionMessage():

Register an Exception message.

void
tcaRegisterExceptionMessage(*msgName*, *msgForm*)
char **msgName*, **msgForm*;

Registers the exception-class message called *msgName*, which has the data format specified in *msgForm*. The data format must be the same as that of the *failureData* parameter used by tcaFailure().

To register the handler for the exception message, use tcaRegisterHandler() (see page 10).

Attaching exception handlers to nodes or messages

Exception handlers can be associated with messages by calling tcaAddExceptionHandlerToMessage():

Add an exception handler to a message.

void
tcaAddExceptionHandlerToMessage(*msgName*, *hndName*)
char **msgName*
char **hndName*;

Associates the exception handler *hndName* with the message *msgName*. Each time the message *msgName* is sent, the handler *hndName* is attached to the tree node corresponding to *msgName*.

Several handlers for the same exception can be attached to a node, and when the exception is raised, TCA tries all handlers one by one, starting from the latest registered one, until a success is reached or the list is exhausted.

If the message is already associated with this handler, then the addition is ignored.

You can also add an exception handler to a specific node in the task tree, by using the procedure described next:

Add an exception handler to a task tree node.

void
tcaAddExceptionHandler(*ref*, *hndName*)
TCA_REF_PTR *ref*;
char **hndName*;

Attaches handler *hndName* to node *ref*. Several handlers for the same exception can be attached to a node, and when the exception is raised, TCA tries all handlers one by one, starting from the latest registered one, until a success is reached or the list is exhausted.

If the task tree node already has this handler then the addition is ignored.

Task trees and the context of exception handling

Exception handling is often context-dependent: the same situation may need to be handled differently, depending on the environment and where in the plan the exception occurs. In TCA, you establish the context of an exception handler by attaching it to a particular task tree node. When an exception is raised, TCA searches up the task tree, starting from the node where the exception was issued, to find a handler specific to that exception. TCA then invokes the handler.

Thus, the scope of a handler is determined by its position in the task tree hierarchy. Exception handlers placed closer to the root node typically have more general effect than those placed near leaves.

Handling exceptions

Typical ways an exception handler can deal with exceptions include:

- issuing queries for sensory data and information about the current environment,
- calling tcaReferenceData() (see page 32) to retrieve data associated with problematic task tree nodes.
- manipulating the task tree by killing subtrees or inserting new nodes that implement error recovery. (See "Operations on Task Trees" on page 30 for tree manipulation routines.)
- calling tcaGetExceptionInfo() to get references to problematic task tree nodes and information such as the name of the exception,
- calling tcaRetry() to try to re-achieve a tree node.

If an exception handler finds it cannot handle the situation, it should call tcaByPassException(). When this procedure is called, TCA resumes the search up the task tree for a capable handler. If no handler is found, TCA will terminate the failed task and all its subtasks, killing the associated task tree branch.

An exception handler can suppress an exception and raise a new one by calling tcaFailure(). For example, the handler may diagnose the exception, distill it, and then raise an abstracted version of the exception, which is to be the input to another handler at a higher level of the task tree.

An exception handler indicates that it has dealt with the problem by calling tcaSuccess() and returning.

Get task tree information about the exception.

void
tcaGetExceptionInfo(*ref*, *info*)
TCA_REF_PTR *ref*;
EXC_INFO_PTR *info*;

```
typedef struct {
    TCA_REF_PTR ERef, CRef;
    int numOfRetries;
} EXC_INFO_TYPE, *EXC_INFO_PTR;
```

Fills the slots of info: ERef, the reference to the exception node; CRef, the reference to the context node; and numOfRetries, the number of times the context node has been retried (see tcaRetry()), respectively. The parameter *ref* is the first argument that was passed to the exception handler.

When an exception handler is invoked, two special task tree nodes are defined: the exception node, corresponding to the exception message raised, and the context node, the node to which the exception handler is attached. These nodes are often useful in determining the problem and repairing it, and this procedure supplies references to both tree nodes as well as other useful information.

Note that *info* must be a pointer to an already allocated EXC_INFO_TYPE record.

Retry a failed tree node.

void
tcaRetry(*ref*)
TCA_REF_PTR *ref*;

Kills the subtree rooted at node *ref*, and then tries to re-achieve the node by re-sending the associated message. The parameter *ref* can be any valid task tree node reference whose corresponding message has already been handled or is currently being handled.

To prevent a node from being repeatedly retried, the number of times the node has been retried can be accessed (see tcaGetExceptionInfo()) and checked to determine whether to stop retrying. Like tcaSuccess(), tcaRetry() should be the last call before the handler routine returns.

Inform TCA server that the current exception handler is unable to handle the situation.

void
tcaByPassException(*ref*)
TCA_REF_PTR *ref*;

ref is the first argument passed to the exception handler. This informs TCA that the exception handler was unable to deal with the situation, and that the next exception handler should be invoked. This procedure can only be called within an exception handler. Like tcaSuccess(), tcaByPassException() should be the last call before the handler routine returns.

An example—docking

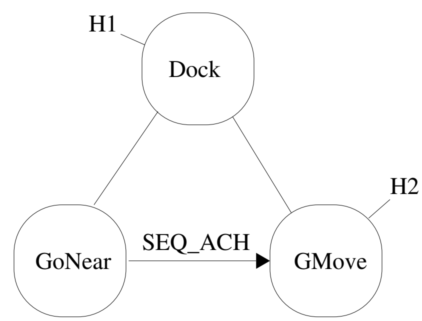
The figure on the right shows part of the task tree for docking a certain robot on its battery charger. If a “stuck wheels” failure is detected from below the “Go Near” node (which could occur if the robot unexpectedly collided with an object), TCA invokes handler “H1”, which handles the exception by retrying the “Go Near” and “GMove” nodes. In actually docking with the charger, however, the robot backs up past where it expects the charger to be in order to get a tight connection. In this context, a “stuck wheels” exception is handled by “H2”, which ignores the exception if it finds that it is in charging mode. If not (i.e., the robot missed the charger), the exception is passed up and “H1” is invoked to retry the docking maneuver.

```
void DockHnd(ref, empty)
TCA_REF_PTR ref; char *empty;
{
    TCA_REF_PTR gRef;
    tcaAddExceptionHandler(ref, “H1”);
    tcaExpandGoal(“Go Near”, Charger_Position);
    gRef = tcaCreateReference (“Gmove”);
    tcaAddExceptionHandler(gRef, “H2”);
    tcaExecuteCommandWithConstraints(gRef, “Gmove”,
            -40.0, SEQ_ACH);
}

void H1(ref, data)
TCA_REF_PTR ref; char *data;
{
    EXC_INFO_TYPE info;
    tcaGetExceptionInfo(ref, &info);
    if (info.numOfRetries < 3)
            tcaRetry(info.CRef);
    else
            tcaFailure(ref, “FailToDock”, NULL);
}

void H2(ref, data)
TCA_REF_PTR ref; char *data;
{
   int isDocking;
   tcaQuery(“docking?”, NULL, &isDocking);
   if (isDocking)
            tcaSuccess(ref);
   else
            tcaByPassException(ref);
}

main()
{
    ...
   tcaRegisterExceptionMessage(“stuck wheels”, NULL);
   tcaRegisterHandler(“stuck wheels”, “H1”, H1);
   tcaRegisterHandler(“stuck wheels”, “H2”, H2);
    ...
}
```



Wiretaps

Suppose that you have developed motor control routines for a robot, and the routines have been tested and proven reliable. Now, you would like to add to your system a vision procedure that takes a picture after each movement command finishes. Ideally, you do not want to tamper with the motor control routines, because they are stable. Nor do you want to tamper with the main program code, which has also proven reliable. It would be best if you could attach to your system some process that monitored movement commands and triggered the vision procedure at the proper time.

TCA provides such functionality through its wiretap facility. A wiretap is a way for one module to receive a copy of a message destined for another module. Because all messages pass through TCA's central server, TCA can monitor the flow of messages in your system and set up wiretaps on particular messages. Suppose that you place a wiretap on a message. When the central server receives this message from some module, the server prepares to deliver a "listening message"—some goal, command, constraint, or monitor message that you have previously defined. When setting up the wiretap, you specify exactly when the listening message should be delivered, relative to the status of the tapped message. (See "Wiretap Conditions" below.)

For example, you may want TCA to issue the message "TakePicture" each time the goal "ShortWalk" has been achieved. That is, a message called "TakePicture" should be delivered to its handler after all commands triggered by a "ShortWalk" goal message have been executed. You would set up a tap on the "ShortWalk" message, specifying "TakePicture" as the listening message and AfterAchieved as the condition for sending "TakePicture." Each time the message "ShortWalk" is sent by some module, the TCA server will prepare to send a "TakePicture" message. The server will deliver the "ShortWalk" message, wait for it to be achieved, and then deliver the "TakePicture" message.

Interfaces that monitor a group of messages and display information about a system can make use of message taps to selectively watch a small set of messages. You could, for example, write a graphical display program that kept track of a robot's movement by tapping any command messages related to movement. The advantage of such an interface is that it could be brought up quickly without affecting an existing system. Furthermore, the system can still function without the interface module.

In fact, the wiretaps are in effect only when the tapping module is running. If the module that requested the wiretap exits (or crashes), TCA will stop sending the wiretaps; when the module is restarted, the wiretaps will be resumed.

The following procedures allow you to set up taps:

Set up a tap on message.

TCA_RETURN_VALUE_TYPE
tcaTapMessage(*condition*, *tappedMsg*, *listeningMsg*)
TAP_CONDITION_TYPE *condition*;
char **tappedMsg*, **listeningMsg*;

Sets up a tap on the message *tappedMsg*. The listening message is sent with the same data as the tapped message, whenever the indicated condition of the tapped message holds. The listening message cannot be a query message. It must be the name of a constraint, goal, command, or monitor message, although different wiretap conditions impose additional constraints on the class of listening message allowed—see "Set up a tap on a given reference." below. Note that tcaTapMessage() applies to all instances of the tapped message. To tap a particular message instance, use tcaTapReference().

For example:

```
tcaTapMessage(AfterAchieved, "ShortWalk", "TakePicture");
```

Note that any messages that have already been sent cannot be tapped.

Set up a tap on a given reference.

TCA_RETURN_VALUE_TYPE
tcaTapReference(*condition*, *tappedRefPtr*, *listeningMsg*)
TAP_CONDITION_TYPE *condition*;
TCA_REF_PTR *tappedRefPtr*;
char **listeningMsg*;

The listening message *listeningMsg* is sent with the same data as that of the *tappedRefPtr*, when the indicated condition of the tapped reference holds. Note that, unlike tcaTapMessage(), tcaTapReference() applies to just a single message. The *tappedRefPtr* must refer to an instance of a goal, command, or monitor message. The listening message must also be a constraint, goal, command, or monitor message.

The effects of a tap are unpredictable if the condition has already occurred when the tap is initiated (e.g., if a condition of "BeforeHandling" is given when the *tappedRefPtr* message has already been handled).

Wiretap Conditions

The condition parameter for tcaTapMessage(), tcaTapReference(), and tcaRemoveTap() is one of the following constants of type TCA_CONDITION_TYPE:

- WhenSent
  Send the listening message as soon as the central server receives the tapped message.

- BeforeHandling
  Deliver the listening message to its handler when the central server is *ready* to deliver the tapped message to its handler (i.e., all temporal constraints have been met) but *before* the delivery takes place. If both the listening and tapped messages are either goal, command, or monitor messages, the listening message temporally precedes the tapped message (i.e., the tapped message is delayed until the listening message is handled).

- WhileHandling
  Deliver the listening message when the central server delivers the tapped message to its handler. The listening message and tapped message may be handled concurrently. Handling of the listening message is not guaranteed to finish before handling of the tapped message finishes.

- AfterHandled
  Send the listening message after the tapped message has been handled. The listening message may be handled concurrently with succeeding messages.

- AfterReplied (Query only)
  Deliver the listening message after the reply to the tapped query message has been sent by its handler. This condition differs from “AfterHandled” in that the listening message is sent the reply data of the tapped message; with the “AfterHandled” condition, it would be sent the original query data.

- AfterSuccess (Goal/Command/Monitor only)
  Send the listening message after the tapped goal, command, or monitor message has indicated “Success”. This is similar to the AfterHandled condition.

- AfterFailure (Goal/Command/Monitor only)
  Send the listening message after the tapped goal, command, or monitor message has indicated “Failure.” This is similar to the AfterHandled condition. Note, that this condition is just included for completeness: the exception handling mechanisms should be used to cover this case.

- BeforeAchieving (Goal/Command/Monitor only)
  Send the listening message when the tapped goal, command, or monitor message is ready to be achieved (according to the temporal constraints). If the listening message is also a goal, command, or monitor, then the achievment of the tapped message will be delayed until the listening message is achieved.

- WhileAchieving (Goal/Command/Monitor only)
  Send the listening message when the tapped goal, command, or monitor message has started to be achieved. The listening message may be achieved concurrently with the achievement of the tapped message. (See BeforeAchieving for contrast.)

- AfterAchieved (Goal/Command/Monitor only)
  Send the listening message after the tapped goal, command, or monitor message (and all its submessages) has been achieved. The listening message does not necessarily have precedence over succeeding messages—see WhenAchieved.

- WhenAchieved (Goal/Command/Monitor only)
  Send the listening message after the tapped goal, command, or monitor message has been achieved. If the listening message is a goal, command, or monitor, its achievement is considered part of the achievement of the tapped message (i.e., the end of the listening message’s achievement interval precedes the end of the tapped message’s achievement interval). This means that succeeding messages whose temporal constraints are related to the achievement of the tapped message will be delayed until the listening message is achieved.

- BeforePlanning (Goal only)
  Similar to “BeforeAchieving”, except based on the planning interval of the tapped message.

- WhilePlanning (Goal only)
  Similar to “WhileAchieving”, except based on the planning interval of the tapped message.

- AfterPlanned (Goal only)
  Similar to “AfterAchieved”, except based on the planning interval of the tapped message.

- WhenPlanned (Goal only)
  Similar to “WhenAchieved”, except based on the planning interval of the tapped message.

- WhenKilled
  Send the listening message when the task tree branch associated with the tapped message is killed. Note that if the tapped message is being handled when its associated task subtree is killed, then TCA allows the message to continue to completion. In such cases, the listening message is sent when the tapped message has actually completed.

Return a reference to message being tapped.

TCA_REF_PTR
tcaFindTappedReference(*listeningRefPtr*)
TCA_REF_PTR *listeningRefPtr*;

Returns a reference to the message that was tapped by the *listeningRefPtr*. Effects are undefined if *listeningRefPtr* refers to a constraint message.

Remove a message tap.

TCA_RETURN_VALUE_TYPE
tcaRemoveTap(*condition*, *tappedMsg*, *listeningMsg*)
TAP_CONDITION_TYPE *condition*;
char **tappedMsg*, **listeningMsg*;

Removes the *listeningMsg*/*condition* pair from the set of taps on the tapped message. Undoes the effect of tcaTapMessage().

TCA and X Windows

A TCA module that uses the **X** window manager needs to determine when a message needs to be processed from the TCA server or when an **X** window event has been triggered. Finding out which event occured is done through a UNIX select() call. If a message from TCA needs to be processed, call tcaHandleMessage(). Events triggered for **X** can be found by calling the **X** window routine XNextEvent().

To make use of select(), one needs to know the appropriate socket numbers. Under **X**11 one can use ConnectionNumber(), defined in X11/Xlib.h, to find the UNIX socket connection to the **X** server. The call **tcaGetServer()** returns the socket number of the TCA central server. A single listenList can then be made by taking the union of these values.

> In some versions of **X**, the include file X.h defines Success, as does TCA in tca.h. To prevent conflicts when including both files, tca.h “undefines” Success.

```
#include <sys/types.h>
#include <X11/Xlib.h>
#include <tca.h>

extern Display *dpy;

void main(void)
{
    int xSocket, tcaSocket;
    fd_set listenList;

     /* initialize X connection */

     FD_ZERO(&listenList);
    xSocket = ConnectionNumber(dpy);
    tcaSocket = tcaGetServer();
     while (1) {
            FD_SET(xSocket, &listenList);
            FD_SET(tcaSocket, &listenList);
            select(FD_SETSIZE, &listenList, 0, 0, NULL);
            if (FD_ISSET(xSocket, &listenList))
                    handleXEvent();
            else if (FD_ISSET(tcaSocket, &listenList))
                    tcaHandleMessage(0);
     }
 }
```

> NOTE: In this example, the function **handleXEvent()** is a user defined function that calls the appropriate X handling functions. In some versions of X, this can be replaced by the call to **notify_dispatch()**.

Get the socket number used by central server.

int
tcaGetServer()

Returns the socket number (file descriptor) of the TCA central server. This number can be used to create a file descriptor mask (via FD_SET) for use in user-specified select() calls. Particularly useful for integrating TCA with other socket-based systems, such as X.

Example programs

This section comprises several small TCA programs taken from a collection of samples used in on-site tutorials. Each of these sample TCA systems is composed of two modules: an “A” module and a “B” module, which are coded in a single files (e.g. “a2.c” and “b2.c”).

Compiling and linking

In your code, you will need to include the file “tca.h.” No other TCA include files are needed.

The following “makefile” specifies how to compile the files in these examples. This particular makefile assumes that a Sun4 is being used, but library files are available for Sun4’s, NeXT machines, and 68030 processors running VxWorks.

```
#
# Makefile for TCA tutorials
#
# Location of the tca top-level directory (this is arbitrary)
TCADIR = /usr/local/tca
# Location of library files and header files
LIBTCA = $(TCADIR)/lib
TCAPATH = $(TCADIR)/include

CFLAGS = -g -I$(TCAPATH)

a2: a2.o
    cc $(CFLAGS) -o a2 a2.o -L$(LIBTCA) -ltca_sun4

b2: b2.o
    cc $(CFLAGS) -o b2 b2.o -L $(LIBTCA) -ltca_sun4

a3: a3.o
    cc $(CFLAGS) -o a3 a3.o -L$(LIBTCA) -ltca_sun4

b3: b3.o
    cc $(CFLAGS) -o b3 b3.o -L $(LIBTCA) -ltca_sun4

# and so on for the rest of the files
.
.
.
```

Running programs

To run any of these examples, you would need three different terminals, or, for instance, three different xterm windows. In one, you would run the central server, specifying that there are two modules in the system:

(Window 1)

```
% hostname
c.gp.cs.cmu.edu

% central 2
```

In the other two windows, you would define the environment variable CENTRALHOST, specifying the computer that is running the central server. Then you could start the modules. The modules can be started in either order, but the central server must be started first. (In the following examples, the host “c.gp.cs.cmu.edu” is running the central server.)

(Window 2)

```
% setenv CENTRALHOST c.gp.cs.cmu.edu
% a2
```

(Window 3)

```
% setenv CENTRALHOST c.gp.cs.cmu.edu
% b2
```

Note that the central server and the modules can be run on the same machine, or on different machines.

Primitive data structures

```
/*------------------------------------------------------------------------
   File: prim.h
 ------------------------------------------------------------------------*/

      typedef int INT_TYPE, *INT_PTR;
      #define INT_FORMAT “int”

      typedef float FLOAT_TYPE, *FLOAT_PTR;
      #define FLOAT_FORMAT “float”

      typedef double DOUBLE_TYPE, *DOUBLE_PTR;
      #define DOUBLE_FORMAT “double”

      typedef char CHAR_TYPE, *CHAR_PTR;
      #define CHAR_FORMAT “char”

      typedef char *STRING_TYPE, **STRING_PTR;
      #define STRING_FORMAT “string”
```

```
/*------------------------------------------------------------------------
   File: a2.c
 ------------------------------------------------------------------------*/

      #include <stdio.h>
      #include “tca.h”
      #include “prim.h”

      main()
      {
      INT_TYPE x;
      FLOAT_TYPE f;
      DOUBLE_TYPE d;
      CHAR_TYPE c;
      STRING_TYPE str;

      printf(“Connect ...\n”);
      tcaConnectModule(“Module A”, tcaServerMachine());

      tcaWaitUntilReady();

      x = 17;
      f = 1.0;
      d = 2.0;
      c = ‘A’;
      str = (STRING_TYPE)malloc(sizeof(char)*4);
      str[0] = ‘a’;
      str[1] = ‘b’;
      str[2] = ‘c’;
      str[3] = ‘\0’;

      tcaExecuteCommand(“intMsg”, &x);
      tcaExecuteCommand(“floatMsg”, &f);
      tcaExecuteCommand(“doubleMsg”, &d);
      tcaExecuteCommand(“charMsg”, &c);
      tcaExecuteCommand(“stringMsg”, &str);
      }
```

```
/*------------------------------------------------------------------------
   File: b2.c
 ------------------------------------------------------------------------*/

   #include <stdio.h>
   #include “tca.h”
   #include “prim.h”

   void intMsgHnd(ref, x)
   TCA_REF_PTR ref;
   INT_PTR x;
   {
      printf(“*x : %d\n”, *x);
      tcaSuccess(ref);
      tcaFreeData(tcaReferenceName(ref), x);
   }


   void floatMsgHnd(ref, f)
   TCA_REF_PTR ref;
   FLOAT_PTR f;
   {
      printf(“*f : %f\n”, *f);
      tcaSuccess(ref);
      tcaFreeData(tcaReferenceName(ref), f);
   }

   void doubleMsgHnd(ref, d)
   TCA_REF_PTR ref;
   DOUBLE_PTR d;
   {
      printf(“*d : %f\n”, *d);
      tcaSuccess(ref);
      tcaFreeData(tcaReferenceName(ref), d);
   }

   void charMsgHnd(ref, c)
   TCA_REF_PTR ref;
   CHAR_PTR c;
   {
      printf(“*c : %c\n”, *c);
      tcaSuccess(ref);
      tcaFreeData(tcaReferenceName(ref), c);
   }

   void stringMsgHnd(ref, str)
   TCA_REF_PTR ref;
   STRING_PTR str;
   {
      printf(“*str: %s\n”, *str);
      tcaSuccess(ref);
      tcaFreeData( tcaReferenceName(ref), str);
   }

   main()
   {
      printf(“Connect ...\n”);

      tcaConnectModule(“Module B”, tcaServerMachine());

      tcaRegisterCommandMessage(“intMsg”, INT_FORMAT);
      tcaRegisterHandler(“intMsg”, “intMsgHnd”, intMsgHnd);

      tcaRegisterCommandMessage(“floatMsg”,
            FLOAT_FORMAT);
      tcaRegisterHandler(“floatMsg”, “floatMsgHnd”,
            floatMsgHnd);

      tcaRegisterCommandMessage(“doubleMsg”,
            DOUBLE_FORMAT);
      tcaRegisterHandler(“doubleMsg”, “doubleMsgHnd”,
            doubleMsgHnd);

      tcaRegisterCommandMessage(“charMsg”,
            CHAR_FORMAT);
      tcaRegisterHandler(“charMsg”, “charMsgHnd”,
            charMsgHnd);

      tcaRegisterCommandMessage(“stringMsg”,
            STRING_FORMAT);
      tcaRegisterHandler(“stringMsg”, “stringMsgHnd”,
            stringMsgHnd);

      tcaWaitUntilReady();

      tcaModuleListen();
}
```

Composite data structures

```
/*---------------------------------------------------------------------------
    File: structs.h
---------------------------------------------------------------------------*/

    typedef struct {
        int x;
        int y;
        int z;
    } STRUCT_1_TYPE, *STRUCT_1_PTR;

    #define STRUCT_1_FORMAT "{int, int, int}"

    /*************************************************/

    typedef struct {
        int x;
        char c;
        float f;
    } STRUCT_2_TYPE, *STRUCT_2_PTR;

    #define STRUCT_2_FORMAT "{int, char, float}"

    /*************************************************/

    typedef struct {
        char c;
        int x;
        float f;
    } STRUCT_3_TYPE, *STRUCT_3_PTR;

    #define STRUCT_3_FORMAT "{char, int, float}"

    /*************************************************/

    typedef struct {
        float f;
        int x;
        char c;
    } STRUCT_4_TYPE, *STRUCT_4_PTR;

    #define STRUCT_4_FORMAT "{float, int, char}"

    /*************************************************/
```

```
/*---------------------------------------------------------------------------
    File: a3.c
---------------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"
    #include "structs.h"

    main()
    {
        STRUCT_1_TYPE struct1;
        STRUCT_2_TYPE struct2;
        STRUCT_3_TYPE struct3;
        STRUCT_4_TYPE struct4;

        printf("Connect ...\n");
        tcaConnectModule("Module A", tcaServerMachine());

        tcaWaitUntilReady();

        struct1.x = 1;
        struct1.y = 2;
        struct1.z = 3;

        struct2.x = 1;
        struct2.c = 'A';
        struct2.f = 2.0;

        struct3.x = 1;
        struct3.c = 'A';
        struct3.f = 2.0;

        struct4.x = 1;
        struct4.c = 'A';
        struct4.f = 2.0;

        tcaExecuteCommand("struct1Msg", &struct1);
        tcaExecuteCommand("struct2Msg", &struct2);
        tcaExecuteCommand("struct3Msg", &struct3);
        tcaExecuteCommand("struct4Msg", &struct4);
    }
```

```
/*-------------------------------------------------------------------------
    File: b3.c
-------------------------------------------------------------------------*/

    #include <stdio.h>
    #include “tca.h”
    #include “structs.h”

    void struct1MsgHnd(ref, s)
    TCA_REF_PTR ref;
    STRUCT_1_PTR s;
    {
        printf(“struct 1\n”);
        printf(“s->x: %d\n”, s->x);
        printf(“s->y: %d\n”, s->y);
        printf(“s->z: %d\n”, s->z);
        tcaSuccess(ref);
        tcaFreeData(tcaReferenceName(ref), s);
    }

    void struct2MsgHnd(ref, s)
    TCA_REF_PTR ref;
    STRUCT_2_PTR s;
    {
        printf(“struct 2\n”);
        printf(“s->x: %d\n”, s->x);
        printf(“s->f: %f\n”, s->f);
        printf(“s->c: %c\n”, s->c);
        tcaSuccess(ref);
        tcaFreeData(tcaReferenceName(ref), s);
    }

    void struct3MsgHnd(ref, s)
    TCA_REF_PTR ref;
    STRUCT_3_PTR s;
    {
        printf(“struct 3\n”);
        printf(“s->x: %d\n”, s->x);
        printf(“s->f: %f\n”, s->f);
        printf(“s->c: %c\n”, s->c);
        tcaSuccess(ref);
        tcaFreeData(tcaReferenceName(ref), s);
    }

void struct4MsgHnd(ref, s)
TCA_REF_PTR ref;
STRUCT_4_PTR s;
{
    printf(“struct 4\n”);
    printf(“s->x: %d\n”, s->x);
    printf(“s->f: %f\n”, s->f);
    printf(“s->c: %c\n”, s->c);
    tcaSuccess(ref);
    tcaFreeData(tcaReferenceName(ref), s);
}

main()
{
    printf(“Connect ...\n”);

    tcaConnectModule(“Module B”, tcaServerMachine());

    tcaRegisterCommandMessage(“struct1Msg”,
            STRUCT_1_FORMAT);
    tcaRegisterHandler(“struct1Msg”, “struct1MsgHnd”,
            struct1MsgHnd);

    tcaRegisterCommandMessage(“struct2Msg”,
            STRUCT_2_FORMAT);
    tcaRegisterHandler(“struct2Msg”, “struct2MsgHnd”,
            struct2MsgHnd);

    tcaRegisterCommandMessage(“struct3Msg”,
            STRUCT_3_FORMAT);
    tcaRegisterHandler(“struct3Msg”, “struct3MsgHnd”,
            struct3MsgHnd);

    tcaRegisterCommandMessage(“struct4Msg”,
            STRUCT_4_FORMAT);
    tcaRegisterHandler(“struct4Msg”, “struct4MsgHnd”,
            struct4MsgHnd);

    tcaWaitUntilReady();

    tcaModuleListen();
}
```

Arrays

```
/*---------------------------------------------------------------------------
    File: arrays.h
----------------------------------------------------------------------------*/

    #define FIXED_ARRAY_DIM1 2
    #define FIXED_ARRAY_DIM2 3

    typedef int
    FIXED_ARRAY_TYPE[FIXED_ARRAY_DIM1][FIXED_ARRAY_DIM2];

    #define FIXED_ARRAY_FORMAT "[int : 2, 3]"

    #define VAR_ARRAY_DIM1 3
    #define VAR_ARRAY_DIM2 2

    /* variable length (2D) array of strings */
    typedef struct {
        int dim1;
        int dim2;
        char **elements;
    } VARIABLE_ARRAY_TYPE;

    #define VARIABLE_ARRAY_FORMAT "{int, int, <string : 1, 2>}"
```

```
/*---------------------------------------------------------------------------
    File: a4.c
----------------------------------------------------------------------------*/
    #include <stdio.h>
    #include "tca.h"
    #include "arrays.h"

    main()
    {
        int i, j, accessor;
        FIXED_ARRAY_TYPE fixArray;
        VARIABLE_ARRAY_TYPE varArray;

        printf("Connect ...\n");
        tcaConnectModule("Module A", tcaServerMachine());
        tcaWaitUntilReady();

        printf("Fixed Array\n");
        for(i=0; i<FIXED_ARRAY_DIM1; i++) {
                for(j=0; j<FIXED_ARRAY_DIM2; j++) {
                        fixArray[i][j] = i+j;
                        printf("%d ", fixArray[i][j]);
                }
                printf("\n");
        }

        printf("Variable Array\n");
        varArray.dim1 = VAR_ARRAY_DIM1;
        varArray.dim2 = VAR_ARRAY_DIM2;
        varArray.elements =
                (char **)malloc(sizeof(char *) * varArray.dim1 *
                varArray.dim2);
        for(i=0; i < varArray.dim1; i++){
                for(j=0; j < varArray.dim2; j++) {
                /* Can't do multiple subscripts on variable
                         length arrays */
                        accessor = i + j*varArray.dim1;
                        varArray.elements[accessor] =
                                (char *)malloc(sizeof(char)*10);
                        sprintf(varArray.elements[accessor],
                                "%d-%d\0", i, j);
                        printf("%s ", varArray.elements[accessor]);
                }
                printf("\n");
        }
        tcaExecuteCommand("fixedArrayMsg", fixArray); /*
                &fixArray is ignored */
        tcaExecuteCommand("varArrayMsg", &varArray);
    }
```

```
/*------------------------------------------------------------------------
   File: b4.c
-------------------------------------------------------------------------*/

   #include <stdio.h>
   #include "tca.h"
   #include "arrays.h"

   void fixedArrayMsgHnd(ref, fixedArray)
   TCA_REF_PTR ref;
   FIXED_ARRAY_TYPE fixedArray;
   {
      int i, j;

      printf("fixedArrayMsgHnd: start.\n");

      for(i=0; i < FIXED_ARRAY_DIM1; i++) {
            for(j=0; j < FIXED_ARRAY_DIM2; j++)
                  printf("%d ", fixedArray[i][j]);
            printf("\n");
      }

      printf("fixedArrayMsgHnd: done.\n");

      tcaSuccess(ref);
      tcaFreeData(tcaReferenceName(ref), fixedArray);
   }

   void varArrayMsgHnd(ref, varArray)
   TCA_REF_PTR ref;
   VARIABLE_ARRAY_TYPE *varArray;
   {
      int i, j, accessor;

      printf("varArrayMsgHnd: start.\n");

      for(i=0; i < varArray->dim1; i++){
            for(j=0; j < varArray->dim2; j++) {
                  /* Can't do multiple subscripts on
                        variable length arrays */
                  accessor = i + j*varArray->dim1;
                  printf("%s ", varArray->elements[accessor]);
            }
            printf("\n");
      }

      printf("varArrayMsgHnd: done.\n");

      tcaSuccess(ref);
      tcaFreeData(tcaReferenceName(ref), varArray);
   }

main()
{
   printf("Connect ...\n");

   tcaConnectModule("Module B", tcaServerMachine());

   tcaRegisterCommandMessage("fixedArrayMsg",
         FIXED_ARRAY_FORMAT);
   tcaRegisterHandler("fixedArrayMsg",
         "fixedArrayMsgHnd", fixedArrayMsgHnd);

   tcaRegisterCommandMessage("varArrayMsg",
         VARIABLE_ARRAY_FORMAT);
   tcaRegisterHandler("varArrayMsg", "varArrayMsgHnd",
         varArrayMsgHnd);

   tcaWaitUntilReady();

   tcaModuleListen();
}
```

Linked lists

```
/*-----------------------------------------------------------------------------
    File: link.h
-----------------------------------------------------------------------------*/

    typedef struct _LINK_TYPE {
        int a;
        int b;
        struct _LINK_TYPE *next;
        char *s;
    } LINK_TYPE, *LINK_PTR;

    #define LINK_FORMAT "{int, int, *!, string}"
/*-----------------------------------------------------------------------------
    File: a5.c
-----------------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"
    #include "link.h"

    main()
    {
        LINK_PTR link;

        printf("Connect ...\n");
        tcaConnectModule("Module A", tcaServerMachine());

        tcaWaitUntilReady();

        link = (LINK_TYPE *)malloc(sizeof(LINK_TYPE));
        link->a = 1;
        link->b = 2;
        link->s = (char *)malloc(sizeof(char)*4);
        link->s[0] = 'a';
        link->s[1] = 'b';
        link->s[2] = 'c';
        link->s[3] = '\0';
        link->next = (LINK_TYPE *)malloc(sizeof(LINK_TYPE));
        link->next->a = 3;
        link->next->b = 4;
        link->next->s = (char *)malloc(sizeof(char)*4);
        link->next->s[0] = 'e';
        link->next->s[1] = 'f';
        link->next->s[2] = 'g';
        link->next->s[3] = '\0';
        link->next->next = NULL;

        tcaExecuteCommand("linkMsg", link);
    }
```

```
/*-----------------------------------------------------------------------------
File: b5.c
-----------------------------------------------------------------------------*/

#include <stdio.h>
#include "tca.h"
#include "link.h"

void linkMsgHnd(ref, link)
TCA_REF_PTR ref;
LINK_PTR link;
{
    LINK_PTR tmp;

    printf("linkMsgHnd: start.\n");

    tmp = link;
    while (tmp) {
            printf("link->a: %d\n", tmp->a);
            printf("link->b: %d\n", tmp->b);
            printf("link->s: %s\n", tmp->s);
            printf("\n");
            tmp = tmp->next;
    }

    printf("linkHnd: done.\n");

    tcaSuccess(ref);
    tcaFreeData(tcaReferenceName(ref), link);
}

main()
{
    printf("Connect ...\n");

    tcaConnectModule("Module B", tcaServerMachine());

    tcaRegisterCommandMessage("linkMsg",
            LINK_FORMAT);
    tcaRegisterHandler("linkMsg", "linkMsgHnd",
            linkMsgHnd);

    tcaWaitUntilReady();

    tcaModuleListen();
}
```

User-defined formats

```
/*------------------------------------------------------------------------
    File: name.h
------------------------------------------------------------------------*/

    typedef struct {
        float x, y, z;
    } POINT_TYPE, *POINT_PTR;

    #define POINT_FORMAT "{float, float, float}"

    #define NEW_NAME_FORMAT "point"

    typedef struct {
        POINT_TYPE p1, p2;
    } TWO_POINTS_TYPE, *TWO_POINTS_PTR;

    #define TWO_POINTS_FORMAT "{point, point}"
/*------------------------------------------------------------------------
    File: a6.c
------------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"
    #include "name.h"

    main()
    {
        TWO_POINTS_TYPE twoPoint;

        printf("Connect ...\n");
        tcaConnectModule("Module A", tcaServerMachine());

        tcaWaitUntilReady();

        twoPoint.p1.x = 1.0;
        twoPoint.p1.y = 2.0;
        twoPoint.p1.z = 3.0;

        twoPoint.p2.x = 4.0;
        twoPoint.p2.y = 5.0;
        twoPoint.p2.z = 6.0;

        tcaExecuteCommand("twoPointMsg", &twoPoint);
    }
```

```
/*------------------------------------------------------------------------
File: b6.c
------------------------------------------------------------------------*/

#include <stdio.h>
#include "tca.h"
#include "name.h"

void twoPointMsgHnd(ref, twoPoint)
TCA_REF_PTR ref;
TWO_POINTS_PTR twoPoint;
{
    printf("twoPointMsgHnd: start.\n");

    printf("twoPoint->p1.x: %f\n", twoPoint->p1.x);
    printf("twoPoint->p1.y: %f\n", twoPoint->p1.y);
    printf("twoPoint->p1.z: %f\n", twoPoint->p1.z);

    printf("twoPoint->p2.x: %f\n", twoPoint->p2.x);
    printf("twoPoint->p2.y: %f\n", twoPoint->p2.y);
    printf("twoPoint->p2.z: %f\n", twoPoint->p2.z);

    printf("twoPointMsgHnd: done.\n");

    tcaSuccess(ref);
    tcaFreeData(tcaReferenceName(ref), twoPoint);
}

main()
{
    printf("Connect ...\n");

    tcaConnectModule("Module B", tcaServerMachine());

    tcaRegisterNamedFormatter(NEW_NAME_FORMAT,
            POINT_FORMAT);

    tcaRegisterCommandMessage("twoPointMsg",
            TWO_POINTS_FORMAT);
    tcaRegisterHandler("twoPointMsg", "twoPointMsgHnd",
            twoPointMsgHnd);

    tcaWaitUntilReady();

    tcaModuleListen();
}
```

Blocking queries

```
/*-----------------------------------------------------------------------
    File: query.h
-----------------------------------------------------------------------*/

    typedef struct {
        int x;
        char *str;
    } SAMPLE_TYPE, *SAMPLE_PTR;

    #define SAMPLE_FORMAT "{int, string}"
```

```
/*-----------------------------------------------------------------------
    File: a7.c
-----------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"
    #include "query.h"

    main()
    {
        SAMPLE_TYPE sample, reply;

        printf("Connect ...\n");
        tcaConnectModule("Module A", tcaServerMachine());

        tcaWaitUntilReady();

        sample.x = 17;
        sample.str = (char *)malloc(sizeof(char)*4);
        sample.str[0] = 'a';
        sample.str[1] = 'b';
        sample.str[2] = 'c';
        sample.str[3] = '\0';

        tcaQuery("queryMsg", &sample, &reply);

        printf("reply.x : %d\n", reply.x);
        printf("reply.str: %s\n", reply.str);
    }
```

```
/*-----------------------------------------------------------------------
    File: b7.c
-----------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"
    #include "query.h"

    void queryMsgHnd(ref, sample)
    TCA_REF_PTR ref;
    SAMPLE_PTR sample;
    {
        SAMPLE_TYPE reply;

        printf("sample->x : %d\n", sample->x);
        printf("sample->str: %s\n", sample->str);

        reply.x = 42;
        reply.str = sample->str;

        tcaReply(ref, &reply);
        tcaFreeData(tcaReferenceName(ref), sample);
    }

    main()
    {
        printf("Connect ...\n");
        tcaConnectModule("Module B", tcaServerMachine());

        tcaRegisterQueryMessage("queryMsg",
                SAMPLE_FORMAT, SAMPLE_FORMAT);

        tcaRegisterHandler("queryMsg", "queryMsgHnd",
                queryMsgHnd);

        tcaWaitUntilReady();

        tcaModuleListen();
    }
```

Non-blocking queries

```
/*------------------------------------------------------------------------
   File: a8.c
------------------------------------------------------------------------*/

   #include <stdio.h>
   #include “tca.h”
   #include “query.h”

   main()
   {
      SAMPLE_TYPE sample, reply;
      TCA_REF_PTR queryRef1, queryRef2;

      printf(“Connect ...\n”);
      tcaConnectModule(“Module A”, tcaServerMachine());

      tcaWaitUntilReady();

      sample.x = 17;
      sample.str = (char *)malloc(sizeof(char)*4);
      sample.str[0] = ‘a’;
      sample.str[1] = ‘b’;
      sample.str[2] = ‘c’;
      sample.str[3] = ‘\0’;

      tcaQuerySend(“queryMsg”, &sample, &queryRef1);
      tcaQuerySend(“queryMsg”, &sample, &queryRef2);

      tcaQueryReceive(queryRef2, &reply);

      printf(“query 2\n”);
      printf(“reply.x : %d\n”, reply.x);
      printf(“reply.str: %s\n”, reply.str);

      tcaQueryReceive(queryRef1, &reply);

      printf(“query 1\n”);
      printf(“reply.x : %d\n”, reply.x);
      printf(“reply.str: %s\n”, reply.str);
   }
```

```
/*------------------------------------------------------------------------
   File: b8.c
------------------------------------------------------------------------*/

   #include <stdio.h>
   #include “tca.h”
   #include “query.h”

   int count;

   void queryMsgHnd(ref, sample)
   TCA_REF_PTR ref;
   SAMPLE_PTR sample;
   {
      SAMPLE_TYPE reply;

      printf(“sample->x : %d\n”, sample->x);
      printf(“sample->str: %s\n”, sample->str);

      reply.x = count++;
      reply.str = sample->str;

      tcaReply(ref, &reply);
      tcaFreeData(tcaReferenceName(ref), sample);
   }

   main()
   {
      count = 0;
      printf(“Connect ...\n”);
      tcaConnectModule(“Module B”, tcaServerMachine());

      tcaRegisterQureyMessage(“queryMsg”,
            SAMPLE_FORMAT, SAMPLE_FORMAT);

      tcaRegisterHandler(“queryMsg”, “queryMsgHnd”,
            queryMsgHnd);

      tcaWaitUntilReady();

      tcaModuleListen();
   }
```

Blocking commands

```
/*------------------------------------------------------------------------
    File: blockingCom.h
------------------------------------------------------------------------*/

    typedef struct {
    int x;
    char *str;
    } SAMPLE_TYPE, *SAMPLE_PTR;

    #define SAMPLE_FORMAT "{int, string}"

/*------------------------------------------------------------------------
    File: a9.c
------------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"
    #include "blockingCom.h"

    main()
    {
        SAMPLE_TYPE sample;
        TCA_RETURN_VALUE_TYPE status;

        printf("Connect ...\n");
        tcaConnectModule("Module A", tcaServerMachine());

        tcaWaitUntilReady();

        sample.x = 17;
        sample.str = (char *)malloc(sizeof(char)*4);
        sample.str[0] = 'a';
        sample.str[1] = 'b';
        sample.str[2] = 'c';
        sample.str[3] = '\0';

        status = tcaWaitForCommand("successMsg", &sample);

        if (status == Success) {
                printf("Success\n");
        } else {
                printf("Failure\n");
        }

        status = tcaWaitForCommand("failureMsg", &sample);

        if (status == Success) {
                printf("Success\n");
        } else {
                printf("Failure\n");
        }
    }
```

```
/*------------------------------------------------------------------------
    File: b9.c
------------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"
    #include "blockingCom.h"

    void successMsgHnd(ref, sample)
    TCA_REF_PTR ref;
    SAMPLE_PTR sample;
    {
        printf("sample->x : %d\n", sample->x);
        printf("sample->str: %s\n", sample->str);

        tcaSuccess(ref);
        tcaFreeData(tcaReferenceName(ref), sample);
    }

    void failureMsgHnd(ref, sample)
    TCA_REF_PTR ref;
    SAMPLE_PTR sample;
    {
        printf("sample->x : %d\n", sample->x);
        printf("sample->str: %s\n", sample->str);

        tcaFailure(ref, "example fail", NULL);
        tcaFreeData(tcaReferenceName(ref), sample);
    }

    main()
    {
        printf("Connect ...\n");
        tcaConnectModule("Module B", tcaServerMachine());

        tcaRegisterCommandMessage("successMsg",
                SAMPLE_FORMAT);
        tcaRegisterHandler("successMsg", "successMsgHnd",
                successMsgHnd);

        tcaRegisterCommandMessage("failureMsg",
                SAMPLE_FORMAT);
        tcaRegisterHandler("failureMsg", "failureMsgHnd",
                failureMsgHnd);

        tcaWaitUntilReady();

        tcaModuleListen();
    }
```

Simple example of handler usage

```
/*------------------------------------------------------------------------
    File: aa.c
------------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"

    main()
    {
        int x, i;
        char *str;

        printf("Connect ...\n");
        tcaConnectModule("Module A", tcaServerMachine());

        tcaWaitUntilReady();

        x = 17;
        str = (char *)malloc(sizeof(char)*4);
        str[0] = 'a';
        str[1] = 'b';
        str[2] = 'c';
        str[3] = '\0';

        tcaExecuteCommand("stringMsg", &str);

        tcaExecuteCommand("intMsg", &x);

        tcaQuery("stringQueryMsg", &str, &i);

        printf("i: %d\n", i);
    }


/*------------------------------------------------------------------------
    File: ba.c
------------------------------------------------------------------------*/

    #include <stdio.h>
    #include "tca.h"

    void stringMsgHnd(ref, s)
    TCA_REF_PTR ref;
    char **s;
    {
        printf("s: %s\n", *s);

        tcaSuccess(ref);
        tcaFreeData(tcaReferenceName(ref), s);
    }

void intMsgHnd(ref, x)
TCA_REF_PTR ref;
int *x;
{
    printf("x: %d\n", *x);

    tcaSuccess(ref);
    tcaFreeData(tcaReferenceName(ref), x);
}


void stringQueryMsgHnd(ref, s)
TCA_REF_PTR ref;
char **s;
{
    int x;

    printf("s: %s\n", *s);

    tcaFreeData(tcaReferenceName(ref), s);

    x = 17;

    tcaReply(ref, &x);
}


main()
{
    printf("Connect ...\n");
    tcaConnectModule("Module B", tcaServerMachine());

    tcaRegisterCommandMessage("stringMsg", "string");
    tcaRegisterHandler("stringMsg", "stringMsgHnd",
            stringMsgHnd);

    tcaRegisterCommandMessage("intMsg", "int");
    tcaRegisterHandler("intMsg", "intMsgHnd", intMsgHnd);

    tcaRegisterQueryMessage("stringQueryMsg", "string",
            "int");
    tcaRegisterHandler("stringQueryMsg",
            "stringQueryMsgHnd", stringQueryMsgHnd);

    tcaWaitUntilReady();

    tcaModuleListen();
}
```

Index

A

achievement
  of goals **26**

action message
  of a monitor **35**

attending set (of a resource) **19**

B

blocking messages **12**

C

capacity of a resource **19**

central server
  connecting to **6**
  getting location of **7**, **47**
  starting **5**

CENTRALHOST **5**

command messages
  blocking **15**
  non-blocking **15**
  registering **14**
  tcaDelayCommand **30**

condition message
  of a monitor **35**

constraint messages
  sending **16**

constraints (temporal) **28**–??

contraint messages
  registering **16**, **17**

D

data formats
  arrays **8**
  linked/recursive **9**
  structures **8**
  TCA-supplied **8–9**
  user-defined **9**

DELAY_PLANNING **28**

demon monitor
  registering **37**

E

exception messages
  handling **41**
  registering **40**
  sending **40**

expansion of goals **26**

expected number of modules **5**

F

format strings **8**

G

goal messages
  registering **26**
  sending **26**

goals
  achievement **26**
  expansion **26**
  relation to task tree **24**

H

handler
  definition **5**

handling messages **10–13**

I

interval monitors
  defaults **38**
  demonl **36**
  options **37**
  polling **36**

intervals (temporal)
  achievement **28**
  handling **27**
  planning **28**

L

listening message (of a wiretap) **43**

locking a resource **20**

M

malloc() **18**

memory management
- for C users **17**
- releasing references **33**

message handler
- definition **5**

messages
- blocking vs. non-blocking **12**
- handling **10–13**
- registering **7–10**
- replying to **13**
- sending **11**

modules
- closing **7**
- connecting to central server **6**
- definition **5**
- expected number of **5**
- pausing until all modules are registered **7**

monitors
- definition **35**
- interval
  - demon **36**
  - polling **36**
- point **35**

N

nodes
- creating by sending messages **25**
- creating without sending messages (empty nodes) **26**
- tcaRootNode() **25**

non-blocking messages **12**

NullReply **13**

P

pending set (of a resource) **19**

PLAN_FIRST **28**

point monitors
- initiating **35**
- registering **35**

polling monitors
- registering **36**

Q

query messages
- blocking **12**
- non-blocking **13**
- registering **7**

R

references
- creating **27**
- releasing **33**

registering data formats **8–10**

registering messages **7–10**
- command **14**
- contraint **16**, **17**
- goal **26**
- query **7**

replying to messages
- Failure **15**
- Success **15**

reserving a resource **21**

resource
- adding a handler to **20**
- definition **19**
- locking **20**
- registering **20**
- reserving **21**
- unlocking **20**

retrying a node **41**

S

sending messages
- blocking command **15**
- blocking query **12**
- constraint **16**
- non-blocking command **15**
- non-blocking query **13**

SEQ_ACH **28**

SEQ_PLANNING **28**

sequential achievment **28**

success
- distinction from achievement **15**

T

task trees
  creating 24
  displaying 32
  pruning 33
  root node 25
  traversing 30–32

TCA procedures
  tcaAchievementOf() 28
  tcaAddChildReference() 27
  tcaAddConstraint() 16
  tcaAddExceptionHandler() 41
  tcaAddExceptionHandlerToMessage() 40
  tcaAddHndToResource() 20
  tcaByPassException() 42
  tcaCancelReservation() 21
  tcaCleanUpAfterAchieved() 33
  tcaClose() 7
  tcaConnectModule() 6
  tcaCreateMonitorOptions() 37
  tcaCreateReference() 27
  tcaDefaultTime() 38
  tcaDisplayTaskTree() 32
  tcaEndOf() 28
  tcaExecuteCommand() 15
  tcaExpandCommandWithConstraints() 29
  tcaExpandGoal() 26
  tcaExpandGoalWithConstraints() 29
  tcaFailure() 15, 40
  tcaFindAnteReferenceByName() 31
  tcaFindChildByName() 31
  tcaFindFailedReference() 32
  tcaFindFirstChild() 31
  tcaFindLastChild() 31
  tcaFindNextChild() 31
  tcaFindParentReference() 31
  tcaFindPreviousChild() 31
  tcaFindTappedReference() 45
  tcaFindTopLevelReference() 31
  tcaFireDemon() 37
  tcaFreeData() 18
  tcaFreeReply() 18
  tcaGetExceptionInfo() 41
  tcaGetServer() 47
  tcaHandleMessage() 11
  tcaHandlingOf() 27
  tcaIntervalMonitor() 38
  tcaIntervalMonitorWithConstraints() 37
  tcaIsSameReference() 32
  tcaKillSubTaskTree() 33
  tcaKillTaskTree() 33
  tcaLockModResource() 20
  tcaLockResource() 20
  tcaModuleListen() 11
  tcaNullReply() 13
  tcaPlanningOf() 28
  tcaPointMonitor() 35
  tcaPointMonitorWithConstraints() 36
  tcaQuery() 12
  tcaQueryReceive() 13
  tcaQuerySend() 13
  tcaReferenceData() 32
  tcaReferenceName() 32
  tcaRegisterCommandMessage() 14
  tcaRegisterConstraintMessage() 16, 17
  tcaRegisterDemonMonitor() 37
  tcaRegisterExceptionMessage() 40
  tcaRegisterFreeMemHnd() 18
  tcaRegisterGoalMessage() 26
  tcaRegisterHandler() 10
  tcaRegisterLengthFormatter() 10
  tcaRegisterMallocHnd() 18
  tcaRegisterNamedFormatter() 9
  tcaRegisterPointMonitor() 35
  tcaRegisterPollingMonitor() 36
  tcaRegisterQueryMessage() 7
  tcaRegisterResource() 20
  tcaReleaseReference() 33
  tcaRemoveTap() 45
  tcaReply() 13
  tcaReserveModResource() 21
  tcaReserveResource() 21
  tcaRetry() 41
  tcaRootNode() 25
  tcaServerMachine() 7, 47
  tcaStartOf() 28
  tcaSuccess() 15
  tcaTapMessage() 43
  tcaTapReference() 43
  tcaTplConstrain() 30
  tcaUnlockResource() 20
  tcaWaitForCommand() 15
  tcaWaitForCommandWithConstraints() 29
  tcaWaitUntilReady() 7

TCA_RETURN_VALUE_TYPE 7

tcaDelayCommand 30

temporal constraints 28–??
  applying ??–26, 29–??
  specifying 29

temporal intervals 27

TPL_ORDER() 30

U

unlocking a resource 20

W

wiretaps
  conditions for sending listening message 44
  listening message 43
  removing 45
  setting up 43

X

X Windows
integrating TCA with **47**